National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

University of Alberta

Implementing ADISSA Transformations in the Metaview Metasystem

by

Jesse Ka-Leung Lee

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Masters of Science

Department of Computing Science

Edmonton, Alberta
Fall 1992

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-77121-6

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Jesse Ka-Leung Lee
TITLE OF THESIS: Implementing ADISSA Transformations in the Metaview Metasystem

DEGREE: Masters of Science
YEAR THIS DEGREE GRANTED: 1992

Permission is hereby granted to UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis not extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) . . . . . .

Permanent Address:
1 Dundas Street, 2/F
Mongkok, Kowloon
Hong Kong

Date: Oct 14, 92

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Implementing ADISSA Transformations in the Metaview Metasystem** submitted by **Jesse Ka-Leung Lee** in partial fulfillment of the requirements for the degree of Masters of Science.

. . . . . .
Supervisor: Dr. P. Sorenson

. . . . . .
Examiner: Dr. J. Hoover

. . . . . .
Examiner: Dr. G. Fisher

. . . . . .
Examiner: Dr. G. Boloix

Date: Oct. 14, 92

To my father and to Amy Wong

Abstract

Environment Transformation Language (ETL) is a part of Metaview, a metasystem based on the EARA (Entity-Attribute-Relationship-Aggregate) model. ETL offers much potential in representing transformation processes between different specification methods (e.g., Data Flow Diagram, Structured Chart). ETL also allows software designers to compute software metrics in Metaview. The objective of the thesis is to investigate the use of ETL as a representation vehicle, to a well-defined transformation method. The special features under investigation are the feasibility of supporting transformations using a metasystem approach and the use of software metrics to guide software transformations. The transformation method chosen is a part of the ADISSA methodology, which is not a metasystem approach to software development and is not developed with software metrics in mind to support software measurement activities. It is a challenging test of the Metaview metasystem approach to software transformations and software metrics. An ETL specification for the major transformational activities of ADISSA is completed, the first such investigation of transformations based on a metasystem approach to software environment support. Based on this investigation, an improved version of the existing ETL and the demonstration of the feasibility and of combining metrics with software transformations are achieved.

Acknowledgements

I would like to thank Dr. Paul G. Sorenson and Dr. Germinal Boloix for their guidance and proof-reading my thesis so many times.

I would also like to thank Lettice Tse, Eric Zhu, Narendra Ravi and Piotr Findeisen of the Software Engineering Research Group for their help and support.

Table of Contents

1 Introduction 1
- 1.1 Transformational Paradigm . . . 3
- 1.2 Use of Metrics . . . 4
- 1.3 Metaview . . . 5
- 1.4 Conclusion and Thesis Overview . . . 7

2 An Overview of the ADISSA Specification Approach 8
- 2.1 ADISSA Methodology . . . 8
  - 2.1.1 Data Flow Diagrams . . . 8
  - 2.1.2 Components of ADISSA . . . 9
  - 2.1.3 Overview of ADISSA Method . . . 11
- 2.2 ADISSA Transformation . . . 13
  - 2.2.1 Finite State Machine . . . 13
  - 2.2.2 ADISSA-Transactions . . . 15
  - 2.2.3 FSM-Transactions . . . 15
    - 2.2.3.1 Defining Interpretations . . . 18
    - 2.2.3.2 Merging interpretations . . . 21
  - 2.2.4 Converting FSM-transactions into FSMs . . . 22
    - 2.2.4.1 FSM-transactions without ANDs . . . 23
    - 2.2.4.2 FSM-transactions with ANDs . . . 26
- 2.3 Summary . . . 30

3 Use of Metrics on ADISSA Methodology 32
- 3.1 Metrics in the DFD Environment . . . 33
- 3.2 Controlling the Complexity of ADISSA-Transactions . . . 33
- 3.3 Controlling the Complexity of FSM-Transactions . . . 35
- 3.4 Metrics in the FSM Environment . . . 36
- 3.5 Summary . . . 38

4 Metaview for Many Specification Environments 39
- 4.1 EARA Model . . . 39
- 4.2 EARA Database State . . . 41
- 4.3 Modeling the DFD and FSM Environments . . . 42
  - 4.3.1 Modeling the DFD Environment . . . 43
  - 4.3.2 Modeling the Finite State Machine Environment . . . 45
- 4.4 Summary . . . 46

5 Proposed Extension/Enhancement to Metaview 47
5.1 ETL . . . 47
5.2 Difficulties of Modeling ADISSA Transformations in ETL . . . 48
5.2.1 Keeping the Source Specification Intact . . . 49
5.2.2 Identification of ADISSA-transactions . . . 49
5.2.3 Working with Multiple Environments . . . 52
5.2.4 Decomposing the ADISSA Transformation into Several Phases . . . 52
5.2.5 Multiple Occurrences of Terminators . . . 52
5.2.6 Capturing Intermediate Structures . . . 53
5.2.7 MAP Function . . . 54
5.3 Definition and Operations of Transformation Aggregate . . . 54
5.3.1 Definition and Properties of Transformation Aggregate . . . 55
5.3.2 Manipulation of Transformation Aggregates . . . 57
5.4 Semantics of MAP_OF Function . . . 59
5.5 Other Enhancements to ETL . . . 60
5.6 Summary . . . 62

6 Representing the ADISSA Transformation and Software Metrics in ETL 63
6.1 Outline of the ETL Specification of ADISSA Transformation . . . 63
6.1.1 Assumptions and Overview of the ETL Specification . . . 63
6.1.2 Construction of ADISSA-transactions . . . 64
6.1.3 Applying the SEC Metrics . . . 67
6.1.4 Refinement to FSM-transactions . . . 68
6.1.5 Create FSMs . . . 69
6.2 ETL Specification of Metrics Support for ADISSA Methodology . . . 70
6.2.1 Metrics in the DFD Environment . . . 70
6.2.2 Metrics in the FSM Environment . . . 72
6.3 Summary . . . 74

7 Conclusions & Future Work 75
7.1 Thesis Summary and Contributions . . . 75
7.2 Future Work . . . 77

Bibliography 82

A BNF Description of the Proposed ETL 85

B ETL Specifications 89
B.1 ETL Specification of the ADISSA Transformation . . . 89

List of Figures

1.1 Metaview Architecture 6

2.1 Data Flow Diagrams Elements 9
2.2 Correspondence between DFDs and menu screens 11
2.3 Components and stages of ADISSA 12
2.4 FSMs defined by Babin *et al.* 14
2.5 An example of DFD 16
2.6 Identification of ADISSA-transactions 16
2.7 An example of ADISSA-transaction 19
2.8 First interpretation 20
2.9 Second interpretation 20
2.10 Adjustment of second interpretation 22
2.11 Resulting FSM-transaction 23
2.12 After converting to an FSM 24
2.13 An FSM-transaction contains ANDs 28
2.14 Transaction graph G_0 28
2.15 After applying the rule S4 29
2.16 After applying the rules S5 and S6 29
2.17 After applying the rules S7–S9 to G_0 30
2.18 After applying the rule S10 30
2.19 Resulting transactions after applying the rules R1 and R2 31

3.1 A Traditional Life Cycle 32
3.2 Process that needs refinement 34
3.3 Before adding a data store between p and q 34
3.4 After adding a data store between p and q 34
3.5 p and q are contained in a cycle 35
3.6 An example of ADISSA-transaction 35
3.7 An FSM 37

4.1 Graphical Presentation of Logical Expression 45

5.1 Example of process explosion 49
5.2 Processes p_2 and p_3 explode into the same DFD-P2 50
5.3 Another example of process explosion 51
5.4 After replacing p_2 and p_3 51
5.5 An example of ADISSA-transaction 55
5.6 Conceptual view of transformation aggregates 56

6.1 Structural view of the ETL specification of the ADISSA transformation . . 66

List of Tables

2.1 Elements and corresponding states/entries 25
2.2 Objects and corresponding actions 25
2.3 Mapping T 25

3.1 Results of SEC metrics 36

6.1 Correspondence of rule sets 65
6.2 Mapping a DFD to an FSM 70

Chapter 1

Introduction

Software transformations can be described in terms of transformations between specifications of various levels of detail. The metasystem approach to software development provides a meta-level definition that provides self-description capabilities for CASE (Computer Aided Software Engineering) tool environments [STM88]. The objective of the thesis is to examine the application of a metasystem approach, based on the EARA (Entity-Aggregate-Relationship-Attribute) model [McA88], to a software transformation method from analysis to design. The special features under investigation are the support for transformations and the use of software metrics to guide transformations using ETL (Environment Transformation Language). ETL is a language to specify transformations between specifications [BST91], define product metrics [BST92b] and specify dynamic aspects of process modeling [BST92a]. The transformation being analyzed is within the ADISSA methodology [Sho88, Bab89, BLS91], which combines structured analysis [DeM78, GS79] with system design, interface design, and data modeling in one complete process. The transformations convert a hierarchical data flow diagram (DFD) in the analysis stage into finite state machines (FSMs) in the design stage. We model these transformations using ETL as the representation vehicle and propose an improved version of the existing ETL. In addition, we analyze some software metrics that can assist the transformations to demonstrate the feasibility and advantages of a metric-driven approach for software transformation.

The problem of transformations has received a lot of attention through formal methods, where refinement and reification are applied during the stages of development. CASE tools, supporting the stored representation of specifications, offer much potential to help during the transformation process. Most of the CASE tools use diagrammatic techniques [MM85] on software objects, to permit the developer to visualize the architecture of the system being developed. These tools typically support a specific technique during certain stages of development. Often there is little or no provision for a smooth transition from one stage of the process to the next.

The ultimate goal of an automated transformation system is to provide an environment in which user requirements are automatically converted into software specifications with minimum human intervention. Such a system would not only reduce significantly the forward development effort, but ideally would provide backward analysis to help during maintenance. We believe the goal of automated transformation can best be accomplished using some formal transformation rules derived from the system development methodology under consideration and guided by the use of software metrics and designer input.

The use of software metrics has been recommended to help manage the development and maintenance of computer software since the mid 1970s. Software metrics are quantitative evaluations of certain attributes of a development project that represent our empirical observations, or intuitive understanding, as well as the relationships between the attributes. Some of the attributes are associated with the overall software development project (e.g., time and cost), and some of the attributes are associated with an individual phase of the software life cycle (e.g., structural complexity of specifications). If we can measure these attributes and express them in numbers, we can gain a good understanding of the progress of the project and hence can keep the project under control. Several measures and metrics have been defined and are widely discussed in the literature, e.g., LOC (Lines of Codes), McCabe's cyclomatic complexity [McC76], Halstead Software Science [Hal77], and information flow metrics [HK81]. These metrics are usually applied in the later stages of the development cycle. As a result, when an area of high complexity in the code is detected and this information is fedback to the designer, it is often too late to take any significant corrective action. Thus, the implementor has to trace back to the analysis/design specifications to make major corrective actions.

The importance of applying software metrics earlier in the software development life cycle [Rom90] is now generally recognized. The earlier we apply measurement to the software development life cycle, the earlier we can make any corrective action, predict source code complexity [HS90] and obtain guidance in decision making. More specifically, metrics provide a mechanism to help guide automated transformation.

The effort to develop CASE tool environments is considerable, as are the efforts to enhance or change a support environment to address the special requirements of a development team or application. To reduce significantly the effort in producing a support environment, researchers have proposed meta-level definition approaches that provide self-description capabilities for CASE tool environments. Metaview [STM88] is a metasystem environment based on the EARA model. The model is simple but powerful enough to describe a large class of CASE support environments, e.g. Data Flow Diagram, Structured Chart, Higher Order Software (HOS) [HZ76] environments. Since all CASE environments developed using Metaview are described using the EARA model, automated transformations between specifications in Metaview are feasible. Specifically, they are represented in ETL, which has as its basis the EARA model description of the source and target environments. Besides the power to describe formal transformation rules [BST91], ETL is also powerful enough to define software metrics [BST92b].

In this thesis we represent the ADISSA transformation method in the Metaview metasystem to demonstrate

1. the power of the metasystem approach to software development;
2. the feasibility of automated transformation;
3. the advantages of a metric-driven approach to guide transformations between specifications.

The ADISSA methodology is chosen because

1. it is a well-defined methodology based on Structured Analysis. Fundamental to the ADISSA methodology is a set of rather complex transformation rules which are well-defined to provide guidelines of converting DFDs to FSMs;

2. it was not proposed in conjunction with a metasystem approach to software development. It is supported by a set of PC-based software engineering tools [SM90];

3. it is not a metric-driven approach.

The initial work of ETL [BST91] offers much potential to specify transformation processes between different specification environments in the Metaview metasystem. Since it has never been reported in the literature that a metasystem approach to automated transformations is feasible, this provides an excellent opportunity to examine the applicability of a transformation approach (i.e., using ETL to specify transformations) in a metasystem context. If the transformation rules of the ADISSA methodology can be represented using ETL, this encourages researchers of Metaview to continue with the metasystem approach of automated transformations using ETL as a representation vehicle.

The importance of specifying software metrics to aid software development is realized by the researchers of Metaview [BST92b]. The initial work of the metasystem approach to software metrics in Metaview was able to express a set of software metrics widely discussed in the literature (e.g. [DeM82]) using ETL. It is not known whether software metrics can coexist with a well-defined transformation method (e.g., ADISSA) in Metaview, in the sense that transformations between specifications can be guided or assisted by software metrics. Since the ADISSA methodology is not a metric-driven approach, it provides a test of the possibility of such a coexistence by describing several metrics that could be used in conjunction with the ADISSA transformation method.

The approach adopted in this thesis is to use the first version of ETL [BST91], refine it if necessary, to represent the ADISSA transformation method, and to describe metrics that can coexist with the ADISSA transformation method. The correctness of the transformations is beyond the scope of this thesis (See Section 1.1). The main concern is whether ADISSA transformation can be expressed in ETL. The results of this investigation are an improved version of the existing ETL and the demonstration of the feasibility and advantages of metric-combining approaches with software transformations.

This chapter is organized as follows. Section 1.1 provides an introduction to the transformational paradigm. Section 1.2 discusses the use of metrics in software system development. Section 1.3 provides an introduction to Metaview. Section 1.4 concludes this chapter and gives an overview of the remaining chapters in the thesis.

1.1 Transformational Paradigm

Most currently popular software development methodologies address the various stages of development life cycle (e.g., waterfall model [Boe76], spiral model [Boe88]), from requirements and design specifications to implementation and maintenance. Most of the methodologies provide semi-formal rules and guidelines that establish the transformations occurring from one phase to the next [YC78, GS79]. These guidelines are not formal enough to guarantee the correctness of the transformations between two forms of specification. One exception is the Cleanroom Development method [MDL87, MLH87, Odo90] which uses functional verification to guarantee consistency and statistical testing to assure software quality [MDC86]. The transformational paradigm takes a more radical approach to address the problem: a formal specification undergoes a sequence of formal transformation steps which eventually yield operation code.

Some previous efforts have studied the automatic transformation from a data flow diagram to structured charts. An automated tool to perform transform analysis is reported in [Sha87] and the construction of an expert system called the Specification-Transformation Expert System is described in [TR88]. Boloix *et al.* [BST91] formalize the transformation from a data flow diagram to a structured chart using ETL, based on structured design guidelines given by transform analysis [Pag88]. Babin *et al.* [BLS91] report a transaction-oriented refinement of structured analysis, within the ADISSA methodology [Sho88], applying the transformational paradigm to the requirements and design phases. We will study the ADISSA method more closely in Chapter 2. Sacha [Sac92] proposes a methodology in which a requirements specification written in PAISLey [Zav82, ZS86] is compiled to the form of a Petri net [Pet81] and the net structures are transformed to meet the requirements of the implementation environment.

In its most general form, a transformation is a relation between two specification schemes [PS83]. A transformation is said to be correct if the specification to be transformed and the transformed specification have the same intended meaning or semantics. A transformation is usually represented by transformation rules/algorithms which are mappings from one scheme to another equivalent scheme, such that an element of the domain and its image under the mapping constitute a correct transformation.

The correctness of a transformation method is difficult to establish. The ADISSA methodology, for example, provides a set of well-defined transformation rules; however, the requirements on the DFDs are never addressed. The semantics of DFDs is not well-defined, even though some researchers have attempted to propose or extend the definition of a DFD [TK91, War86]. As a result, the correctness of the ADISSA transformation method is difficult to prove without well-defined semantics. However, Metaview offers the potential to establish the correctness of a transformation. Metaview provides a way to specify the semantics of an environment via Environment Constraint Language (ECL) [McA88]. Pre-conditions and post-conditions of ETL specifications can be specified using ECL to impose constraints on the transforming and transformed specifications to establish the correctness. Undoubtedly, additional work beyond this thesis is required to investigate the feasibility of using ECL to prove the correctness of ETL specifications.

1.2 Use of Metrics

The subject area of software measurement covers a diverse range of topics, such as cost estimation models, productivity models, reliability models, and structural and complexity metrics [CDS86]. Bush and Fenton [BF90] present a conceptual framework that gives a rationalized overview of the subject of software measurement. They propose that the important characteristics of the software development process amenable to measurement are related to processes, products, and resources. Processes are any activities during software development; products are the artifacts that arise out of these processes; and resources are items used during the execution of these processes, as well as the actual performers of these processes. For the purpose of this thesis, we restrict our discussion of the use of metrics to products. Products that are amenable to measurement include analysis/design specifications and source code. Data collection for measurements of these products can be done automatically because these products are usually stored in the CASE repository. Software metrics that are defined on products are usually concerned with size and complexity.

Taking the ADISSA methodology as an example, the analysis specification is represented as a hierarchical DFD and the design specification is represented as a series of FSMs. There are already some software metrics defined for DFDs [DeM82], e.g., determining the number of process in a DFD, determining the deepest level of process explosion, etc. ADISSA is a transaction-oriented methodology where a transaction is represented as a DFD in which all elementary processes are chained to each other via data flows. A process is elementary if it is not exploded into a more detailed DFD. It is desirable to control the size and complexity of a transaction before proceeding with the transformation into FSMs because it is beneficial to apply software metrics as earlier as possible in the development life cycle. These transactions will go through a lot of modifications before they are converted into FSMs. Moreover, designer input is needed to carry out the transformation. In this case, metrics can assist the designer in decision making when supplying the input. After a transformation, it is desirable to control the size and complexity of each finite state machine so that FSMs with high complexity can be fedback to the designer for a more detailed design review.

1.3 Metaview

The EARA metasystem model was proposed by McAllister [McA88, STM88] in an effort to support multiple specification environments. The main goal of Metaview is to generate automatically the major parts of a particular software development environment. The benefits of producing a metasystem is comparable to the use of compiler-compiler systems in the development of compilers today. Figure 1.1 provides a high-level architectural overview of the Metaview metasystem. There are three levels to the architecture: the meta level, environment level and user level.

At the meta level, the *meta definer* defines a meta model which should be sufficiently expressive to support the description of a large class of specification development environments. In our case the meta model is the EARA model. The meta definer should also oversee the activities of creating the database engine, the environment generator, and the tool set facilities.

At the environment level, the *environment definer* uses the environment-model-processor software provided at the meta level to process an environment definition. The environment definition includes an environment model definition, plus tool-specific information on the environment for each generic tool. The result is a database engine and a tool set configured to the required specification environment.

The environment model definition is expressed as an environment definition language (EDL) program, which is processed by the EDL compiler to produce a set of tables. Each tool in the environment is configured using the environment tables and any tool-specific information for the environment.

At the user level, analysts select a particular specification environment. This environment includes a generalized environment model analyzer and the set of configured database engine routines and environment tools. These facilities are used to store and analyze requirement specifications for any information processing system the analyst wants to define. Each specification is stored in the specification database.

Besides EDL, Metaview contains two other languages: ECL (Environment Constraint Language) and ETL (Environment Transformation Language). ECL is used to specify constraints because a model definition of an environment is not sufficiently descriptive

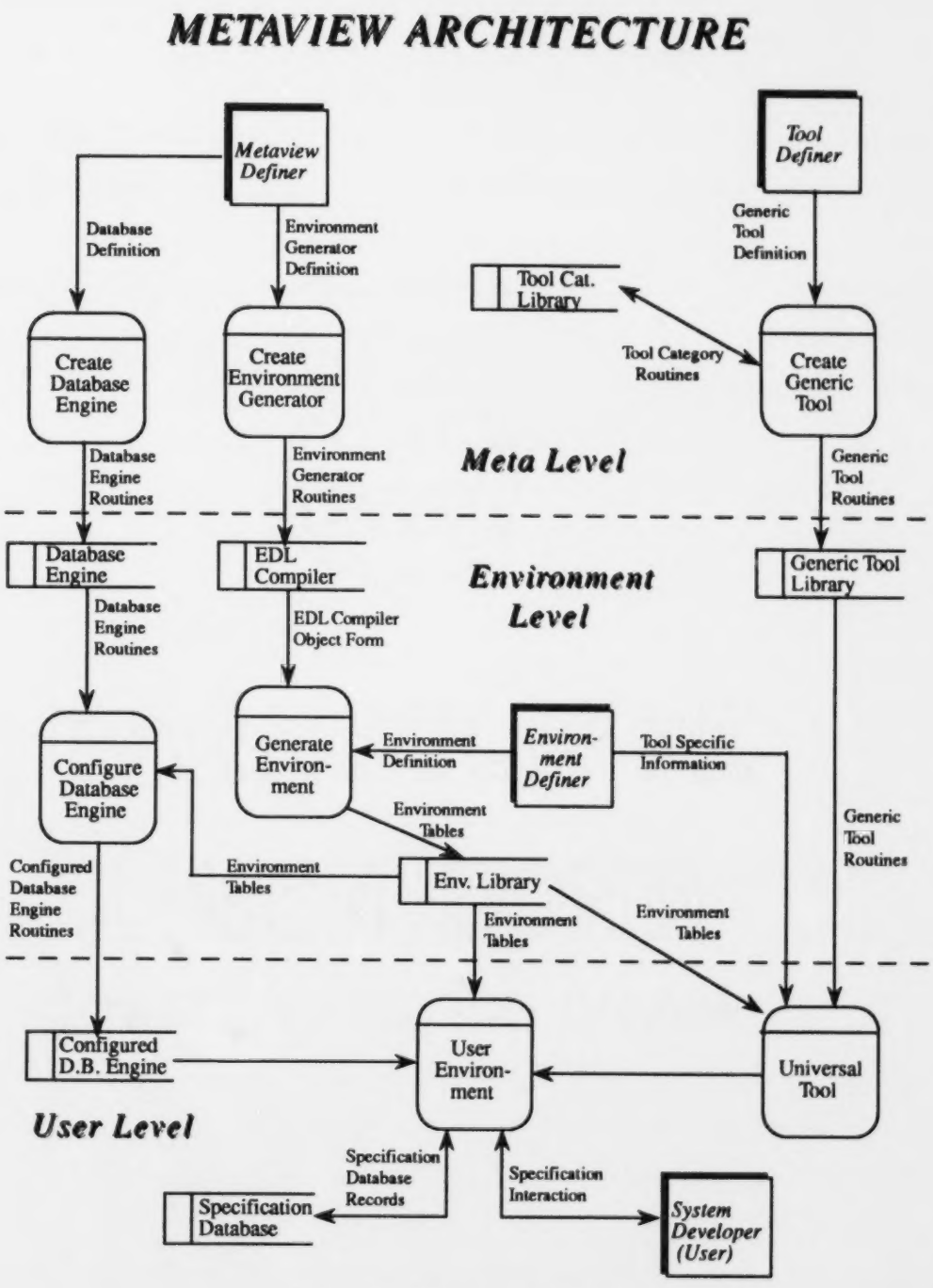


Figure 1.1: Metaview Architecture

to address all issues of consistency and completeness in a specification or methodological restrictions of a particular approach to software development. ETL is used to specify transformations between specifications in Metaview, software metrics and dynamic aspects of process modeling.

1.4 Conclusion and Thesis Overview

Automated transformation in software development is gaining importance because it can reduce significantly the development effort and can guarantee consistency between the output of a phase and its input. It can best be accomplished using some formal transformation rules derived from the system methodology under consideration and guided by the use of software metrics in decision making.

The objective of this thesis is to represent the transformation of DFD to FSM within the ADISSA methodology in Metaview, a metasystem environment based on the EARA model, and to examine the use of software metrics that can assist the ADISSA transformation method. The results are an improved version of the existing ETL and demonstrates the feasibility and advantages of metric-driven approach for software transformation.

The rest of the thesis is organized as follows. Chapter 2 provides a review of the ADISSA methodology and a detailed description of the transformation within ADISSA. Chapter 3 discusses how metrics can be applied to assist the transformations. Chapter 4 provides a review of the EARA model and the concept of EARA database state. We will model the data flow diagram environment and finite state machine environment which are used in the transformations. Chapter 5 discusses difficulties in modeling the transformations in ETL and proposes some enhancements to the existing ETL. These include the definition of a transformation aggregate, the clarification of the semantics of the mapping function and other improvements. Chapter 6 provides an ETL specification of the transformations, and an ETL specification of the metrics discussed in Chapter 3. Chapter 7 concludes the thesis by summarizing the major contributions and suggesting areas for future research.

Chapter 2

An Overview of the ADISSA Specification Approach

In this chapter, we study the ADISSA methodology. Our summary of the methodology is taken primarily from [Sho88, Bab89, BLS91]. ADISSA is a methodology for Architectural Design of Information Systems and Software based on Structured Analysis (SA). It integrates the stages of system analysis and design into one complete process. It differs from the Structured Analysis/Structured Design (SA/SD) [DeM78, GS79] in that instead of Structured Charts (SC) finite state machines (FSMs) are generated as a design specification. Hierarchical data flow diagrams (DFDs) are used in the analysis stage as in SA/SD. From these DFDs, a tree-structure menu is constructed which interfaces between the users and the system. Various types of events are identified and new objects are incorporated into the DFDs. Transactions are then identified to enable a smooth transition from analysis to design. The process of transaction design is formalized by converting each transaction into FSMs which describe the process logic in a rigorous way. The menu can be viewed as the external architecture of the system, while the transactions can be viewed as the internal architecture of the system. Besides the menu and the transactions, the other products of the ADISSA methodology include the input/output schema design and the database schema design.

The chapter is organized as follows. Section 2.1 gives an introduction to the ADISSA methodology. Section 2.2 provides the details of the automated transformation from a DFD to FSMs which is the core of this chapter. The final section summarizes the chapter.

2.1 ADISSA Methodology

2.1.1 Data Flow Diagrams

In ADISSA, DFDs are used to represent the analysis phase as well as the design phase. Since DFDs play an important role in the ADISSA methodology, this section gives a short introduction to DFDs and their refinement in ADISSA methodology. Data flow diagrams have the advantage of reducing the importance of procedural details, emphasizing the flow of data in structuring the eventual design. A DFD is also a description tool for top-down analysis. We can use DFDs to provide both high-level and more detailed views of a system. What takes place within one process on a DFD can be shown in detail in another DFD.

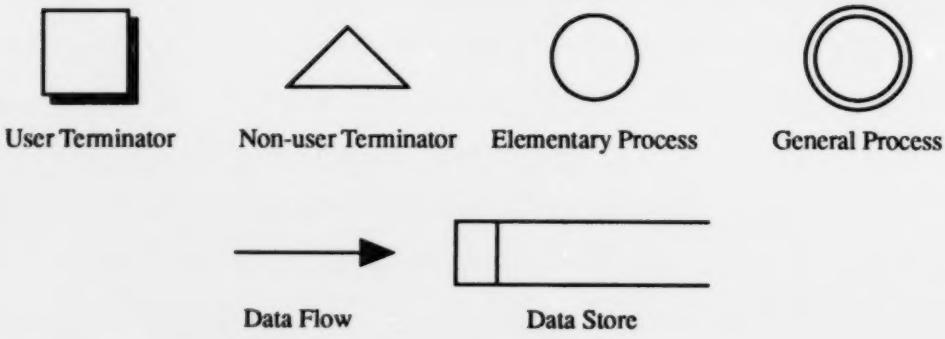


Figure 2.1: Data Flow Diagrams Elements

A data flow diagram is built from four basic components: the data flow, the process, the data store and the terminator. Data originate in several terminators outside the system and flow in through specified paths to processes that transform the data in various ways. Data may be stored temporarily or permanently, but eventually the data finds their way out of the system to the terminators. A process is *elementary* if it is not exploded into a more detailed DFD; otherwise it is *general*.

The ADISSA methodology adopts the current use of DFDs plus the following extensions:

1. Different types of terminators are distinguished.

2. Output data flows of a process are modeled by logical expressions.

The concept of terminator is refined by distinguishing various types of terminators. The most common type of terminators is the *user terminator* (UT), which represents a user in the real world. The non-user terminators are *time terminator* (TT), *real-time terminator* (RTT) and *communication terminator* (CT) and they are used to define non-user transactions. The concept of transaction is discussed in Section 2.1.2.

The output data flows of a process are modeled by logical expressions using the conjunction (AND), disjunction (OR), and exclusive disjunction (XOR) operators.

Even though different types of terminators are distinguished in the ADISSA methodology and logical expressions are added to specify the output data flows of a process, the basic idea of a DFD remains unchanged. Throughout this thesis, we will use the notations shown in Figure 2.1 to draw DFD diagrams.

2.1.2 Components of ADISSA

In this section we describe two important components of ADISSA: *menus* and *transactions*.

Menus

The menu tree, from a user's point of view, represents the external architecture of the system. It forms the interface between the users and the system. Menus are derived from DFDs. An initial menu tree is derived automatically by reviewing the DFD hierarchy in a search of processes connected to user terminators (UTs). Such a DFD leads to the definition of a menu screen that includes a line for every process linked to or from a user terminator (UT). In a menu screen there may be lines which are expandable and others which are not.

A *selection line* causes a more specific menu screen to be presented, and a *terminal line* activates specific procedures in the system. More precisely, a menu includes a terminal line if the user terminator is connected to an elementary process, and a selection line if the user terminator is connected to a general process. The initial menu tree is then improved and refined, taking into consideration factors other than functional organization. Refinement of menus will be discussed in Section 2.1.3.

Transactions
Transactions can be viewed as the internal architecture of the system. Transactions are composed of various processes which are activated in response to user needs and various events in the universe of discourse. ADISSA regards an information system as performing transactions. From a user's point of view, a transaction is an independent task or activity that the system can perform. More precisely, a transaction

1. has a beginning
2. performs a task
3. stops and leaves the system in a coherent state, in the database-transaction sense.

In DFD terms, *a transaction consists of elementary processes which are directly chained to each other via data flows and of data stores and terminators which are connected to these processes.*

A transaction in the system is initiated, or triggered, by an *event*. There are five types of events in ADISSA: (a) *user event*, (b) *time event*, (c) *real-time event* (d) *communication event* and (e) *internal event*. A user event is caused by a user, and the other events are initiated by the system. In DFD terms, a user-transaction contains a user terminator (UT) while non-user transactions contain non-user terminators: TTs, RTTs and CTs. Figure 2.2 depicts a schematic example of menu trees and user-transactions.

Example 2.1 In DFD-F (Figure 2.2), $f1$ is a general process and $f2$ is an elementary process. Since both $f1$ and $f2$ are connected to UTs, there should be two lines in the respective menu screen F. Line (1) in the menu refers to data flow a and line (2) refers to data flow e. Hence, line (1) of the menu is a selection line, whose selection creates another, more specific menu screen to be presented, whereas line (2) is a "terminal line", whose selection initiates a transaction. That transaction involves (as shown in DFD-F) the input of data from UT2, performing procedure $f2$ and updating data store D2. Selection of line (1) in menu screen F would cause the more specific menu screen F1 to be presented. Since the general process $f1$ was decomposed in DFD-F.1 into two elementary processes $f1.1$ and $f1.2$, each connected to a UT, the menu screen F1 has two terminal lines, the selection of either one starting a transaction. If line (1.1) is selected, one possible interpretation of the activated transaction could be: user UT1 inputs data; additional data are retrieved from data store D1; process $f1.1$ resumes execution; and finally data store D3 is updated. If line (1.2) is selected, another transaction is fired. Note that a distinction is made between data flows connecting UTs with general processes and data flows connecting UTs with elementary processes. In the former, data flow a does not indicate a specific transfer of data elements between a UT and a process in the system, but an aggregate of two data flows $a1$ and $a2$. Data flow $a1$ or $a2$ signifies a transfer of data elements between a UT and a elementary process.

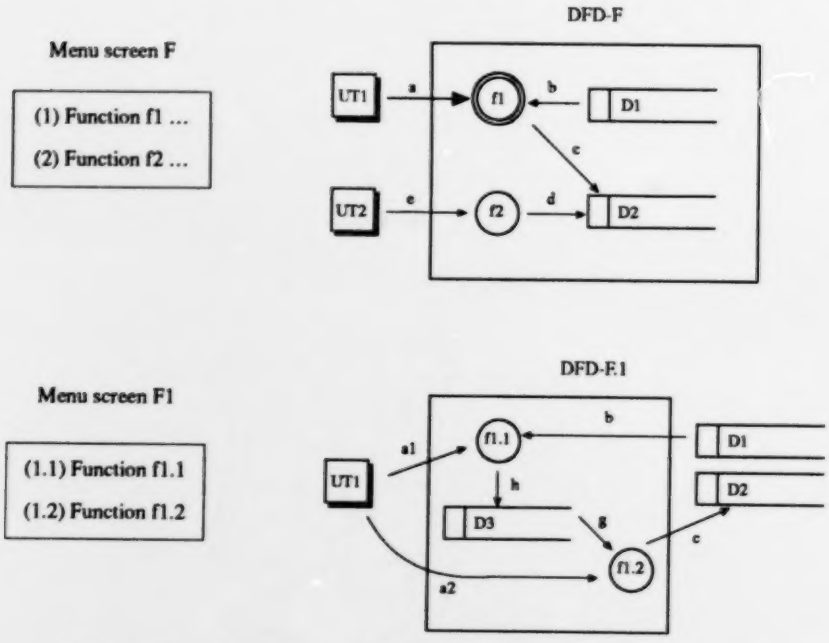


Figure 2.2: Correspondence between DFDs and menu screens

2.1.3 Overview of ADISSA Method

Figure 2.3 (copied from [Sho88]) provides an overview of ADISSA combined with SA, and includes seven stages. Dotted arcs indicate that there is some feedback from Stage (3) to Stage (1) and to Stage (2). It means that the transaction design may influence a change in the specification and the initial menu tree. Stage (5) to Stage (7) create other products: database schema, the input and output schemes, and the data-dictionary (DD) of the system. For the purpose of this thesis, the important stages are Stage (1) to Stage (4).

Stage (1) *functional analysis* is based on the use of hierarchical DFDs. Moreover events are identified and new terminators (TTs, RTTs and CTs) are incorporated within the DFDs and documented in the data-dictionary.

In Stage (2), an initial menu tree is designed as described earlier in Section 2.1.2.

Stage (3) includes the identification of the transactions and of their components; in particular, the linked elementary processes, the terminators and the data stores linked to them. The menu tree is finalized based on the functional structure of the system. All user-transactions will be shown in the menu tree. Finally, the menu tree is refined by removing any degenerate or redundant menus. A degenerate menu occurs when there is only one terminal line in the menu screen. A redundant menu occurs when two lines in the menu screen refer to the same transaction.

In Stage (4), the process logic of a transaction is formalized: a two-level method is employed to describe transactions. The top level describes the "skeleton" of the transaction using English-like notation [Sho88]. The bottom level details the components in a structured description (e.g., pseudo-code for elementary processes). The description of a transaction is not formal enough for automatic transformation. This will be formalized using finite state

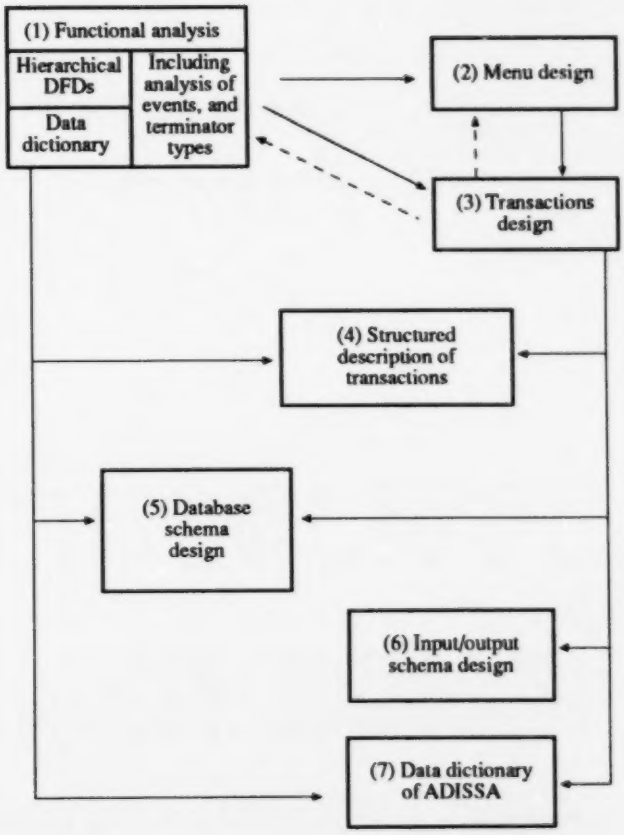


Figure 2.3: Components and stages of ADISSA

machines (Section 2.2).

Example 2.2 We use DFD-F.1 in Figure 2.2 to illustrate the top level description of a transaction. The following is a possible top level description of the transaction which contains the process *f*1.1:

Begin user transaction "1.1"
    Input from UT1: a1
    Read from D1: b
    Execute Process f1.1
    Write to D3: h
End transaction.

2.2 ADISSA Transformation

We use the term *ADISSA transformation* to refer to the transformational approach adopted by Babin *et al.* in [Bab89, BLS91] in which Stage (3) *transaction design* and Stage (4) *structured description of transactions* (Figure 2.3) are formalized.

The transformation process can be divided into three stages:

1. Identification of ADISSA-transactions.
2. Converting ADISSA-transactions into FSM-transactions.
3. Converting FSM-transactions into FSMs.

An ADISSA-transaction identifies the components that a transaction consists of, and an FSM-transaction formally describes the flow of control of an ADISSA-transaction.

Since finite state machines (FSMs) are used to provide a formal description of a transaction in Stage (4), Section 2.2.1 gives a short description of an FSM. Section 2.2.2 gives the details of identifying ADISSA-transactions. Section 2.2.3 describes the rules for converting ADISSA-transactions into FSM-transactions. Section 2.2.4 gives the rules for converting FSM-transactions into FSMs.

2.2.1 Finite State Machine

A finite state machine (FSM) is a mathematical model of a system, with discrete inputs and outputs [HU79]. The FSM can be in any one of a finite number of states. It is assumed that the inputs reach the machine one at a time. Given the current input and current state of the machine, the machine will move to a new state and produce an output. It is possible that the set of inputs and the set of outputs are different. This type of FSM is often known as a *finite automaton*, or more precisely, the *Mealy machine* [HU79].

An FSM, defined by Babin *et al.* [BLS91] for transaction design, is an extension of the basic idea described in the previous paragraph. It contains a finite set of states and a finite set of actions. Moreover, the set of inputs is the same as the set of outputs. To follow the terminology used in [BLS91], we use the term *entry* to refer to an input or an output. Given the current entry and the current state of the machine, the machine will move to a new state and execute one of the elements in the set of actions. This action element takes data from the outside of the FSM and the current entry, processes them, and produces data

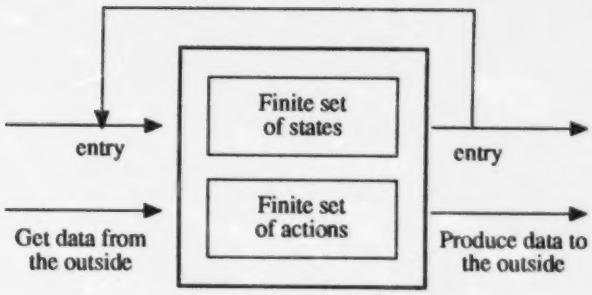


Figure 2.4: FSMs defined by Babin *et al.*

to the outside and the next entry to the machine [BLS91] (see Figure 2.4). "Data from the outside" could mean any information stored in some database which is not part of the FSM. Because an FSM only knows it current state and entry, it does not know, or *remember*, the previous state and entry. Therefore, data from the outside could be the history of all the previous states, the history of all the previous entries, or some special information stored in some database.

Throughout this thesis, we use the term FSM to refer to the FSM defined by Babin *et al.* The following is the formal definition of an FSM defined in [BLS91]. A finite state machine is a system $(S, E, A, T, s_0, e_0, F)$, where

1. S is a nonempty finite set of states.
2. E is a finite set of entries (ν, the empty entry, is a member of E).
3. A is a finite set of actions. An action $a \in A$ takes data from the outside of the FSM and the current entry, processes them and produces data to the outside and the next entry to the machine. It may also produce no entry, in which case, the result of the action is denoted by ν.
4. T is a transformation of $S \times E \rightarrow S \times A$. For the couple (s, e), $T(s, e) = (s', a)$, i.e., on reading entry e, the FSM in state s moves to state s' and executes action a. If the entry is ν, the machine stops.
5. $s_0 \in S$ is the initial state.
6. $e_0 \in E$ is the initial input entry.
7. F, which is a subset of S, is the set of final states.

The FSM accepts the sequence $e_0, e_1, \ldots, e_n$ of entries if and only if:

1. n is finite;
2. The FSM is initially in state s_0 and receiving e_0, has a succession of transitions, and when reading entry e_n, stops in a state $s \in F$.

FSMs are used as design primitives by Babin *et al.* because FSMs are mathematically well-defined and provide a firm basis for proving the consistency between a specification and a design for the control part of the transactions in the ADISSA methodology. A complex

transaction can be represented as a collection of finite state machines. This gives a precise way to draw and conceptualize complex transactions and to check that all possible state transitions have been considered.

2.2.2 ADISSA-Transactions

Identifying an ADISSA-transaction can be done automatically without any human intervention. Let P be the set of *elementary* processes in the DFD. Let us partition the set P by defining an equivalence relation on P. If $p_1, p_2 \in P$, then we define $p_1 \equiv p_2$ if and only if there exists a data flow connecting p_1 and p_2, i.e., either p_1 sends data to p_2 or p_2 sends data to p_1. Each equivalence class contains those processes that are connected to each other via data flows.

For each equivalence class C, we construct an ADISSA-transaction A by doing the following:

(B1) Every process in C is part of A.

(B2) If a data flow is connecting two processes in C, add the data flow to A.

(B3) If a terminator is connected to one of the processes in C, add to A the terminator plus the data flow that connects the process and the terminator.

(B4) If a data store is connected to one of the processes in C, add to A the data store plus the data flow that connects the data store and the process.

After constructing the ADISSA-transaction, if A does not contain any terminator, the original DFD must be modified by adding non-user terminators (TTs, RTTs and CTs), because every transaction in ADISSA must be triggered by some terminator. This is the feedback from Stage (3) to Stage (1) as shown in Figure 2.3.

An ADISSA-transaction created at this stage can be considered as a separate DFD. It represents an individual task that the system can perform.

Example 2.3 We illustrate the identification of ADISSA-transactions using an example. Suppose the original DFD is represented as Figure 2.5. After applying the equivalence relation on P, there are four equivalence classes: $C_1 = \{f5, f6, f7\}$, $C_2 = \{f1\}$, $C_3 = \{f2\}$ and $C_4 = \{f3, f4\}$. For each C_i $(1 \leq i \leq 4)$, we construct an ADISSA-transaction A_i. We add to A_i the initial processes of C_i as well as the data flows, the data stores and the terminators that are directly connected to them. Note that after expansion, A_4 does not contain any terminator. In such a case, we should go back to the original DFD and consider what terminator should trigger this job. Let us assume a time terminator (TT) triggers $f3$. The resulting ADISSA-transactions are illustrated in Figure 2.6.

2.2.3 FSM-Transactions

Identification of ADISSA-transactions enables the interpreter to focus on an independent task that the system is to perform. By an interpreter we mean any person or automated program that is performing a transformation to the DFD.

In the second step of the transformation, as identified at the beginning of Section 2.2, the interpreter converts each ADISSA-transaction into an FSM-transaction.

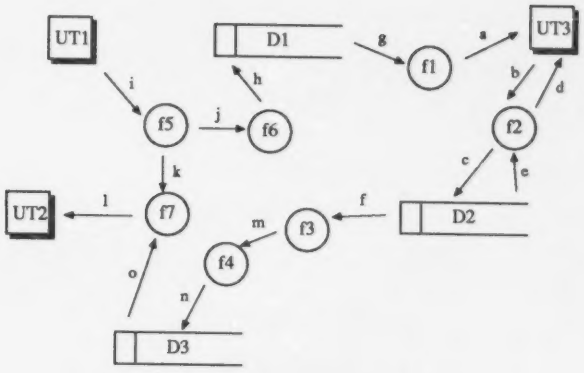
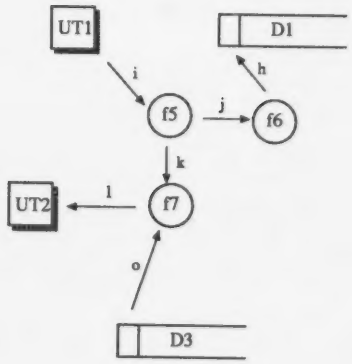
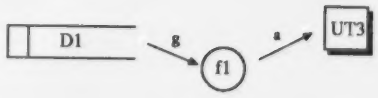


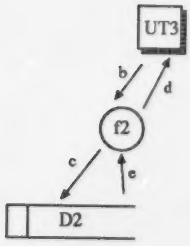
Figure 2.5: An example of DFD



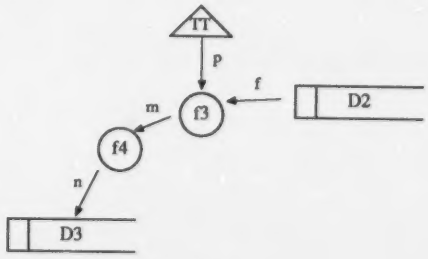
(a) A_1



(b) A_2



(c) A_3



(d) A_4

Figure 2.6: Identification of ADISSA-transactions

Before we give the formal definition of an FSM-transaction, we introduce three new notions:

1. Control data flow (CDF) – Data flows are further classified into two different kinds: data flows which carry the control flow as well and those which do not. A data flow of the first kind is called a CDF while a data flow of the latter kind remains just data flow (DF). A CDF is directed from the triggering to the triggered element, and is

   (a) acting as a trigger for an elementary process or for a terminator, or

   (b) indicating the end of the transaction.

   If the flow of control is in the opposite direction of the flow of data, extra CDFs may be added indicating the direction.

   Example 2.4 Consider Figure 2.6(b). There are two data flows a and g. Data flow g is retrieved from the data store D1 by the process $f1$, and is sent to the terminator UT3 in data flow a. Since the terminator UT3 triggers the process $f1$, the flow of control is in the opposite direction of a. Therefore, an extra CDF is needed and is directed from UT3 to $f1$. Data flow a is a CDF because it indicates the end of the transaction while data flow g remains data flow because neither D1 triggers $f1$ nor $f1$ triggers D1.

2. Starting set – The set of all terminators that trigger the task.

3. Ending set – If the transaction ends on a terminator or on an update of a data store, the terminator or the data store belongs to the ending set.

An FSM-transaction is a formal description of the flow of control and data of an ADISSA-transaction. The components of an FSM-transaction come from the corresponding ADISSA-transaction. In addition, CDFs, starting set, ending set and logical expressions of the outgoing data flows of processes (see Section 2.1.1) are specified during the construction of an FSM-transaction. Elementary processes, terminators and data stores of an FSM-transaction are referred to as *objects*[1] [BLS91].

Definition of FSM-transaction [BLS91]
An FSM-transaction is formed by a set of objects, DFs and CDFs, and if it is constructed from a valid ADISSA-transaction and satisfies the following constraints:

(C1) Any flow connecting two processes, or a process and a terminator, is a CDF.

(C2) Every process can be reached by at least one oriented path of CDFs starting at an object of the starting set.

(C3) Every path of CDFs must end at an object of the ending set.

(C4) A terminator has at most one outgoing CDF.

(C5) CDFs and DFs cannot link more than two objects.

[1] DFs and CDFs are not regarded as objects.

(C6) Only CDFs outgoing from a process may be involved in a Boolean AND.

The rules for construction of an FSM-transaction from a ADISSA-transaction are given in Section 2.2.3.1 and Section 2.2.3.2.

2.2.3.1 Defining Interpretations

The process logic[2] of an ADISSA-transaction is not shown in the DFD and needs to be supplied separately. Because data flows do not provide information on the flow of control, an ADISSA-transaction may be interpreted in more than one way. Relevant interpretations have to be specified by the designer and become part of the transaction's specification. Interpretations are considered as *subtasks* that an ADISSA-transaction is to perform.

Each interpretation does not have to contain all the components of the corresponding ADISSA-transaction. For each interpretation, we generate a separate DFD.

After the designer has provided the information of all the interpretations (subtasks) of the original ADISSA-transaction, the interpreter has to do the following *for each* interpretation (from [BLS91]):

(I1) Define a starting set.

(I2) Define an ending set.

(I3) Make relations between outgoing flows explicit (AND-ed, OR-ed, XOR-ed) for each process. If, according to the designer, several flows outgoing from an object are always used together, they are linked by an AND; if they are used together only in some executions, they are linked by an OR. In all other cases an exclusive OR (XOR) is implied but not written explicitly on the diagram.

(I4) Specification of CDFs and DFs:

1. Any flow connecting two processes, or a process and a terminator, is converted to a CDF.
2. If a flow is from a process to a data store, then:
   (a) if it is not involved in an AND, it is converted to a CDF.
   (b) if it is involved in an AND with flows going only to data stores, it is converted to a CDF.
3. All other flows remain DFs.

If after these transformations an AND involves only DFs, it is removed; if an AND involves both CDFs and DFs, the AND is modified to involve only CDFs. As a result, AND conditions would involve only CDFs. (This is to comply with constraint C6 mentioned in Section 2.2.3.)

(I5) Addition of CDFs: If, according to the designer, the flow of control on a data flow is opposite to the original data flow, an additional CDF is created opposite to the DF. (See Example 2.4.)

[2]The notion of process logic is discussed in Stage (4) of the ADISSA methodology in Section 2.1.3.

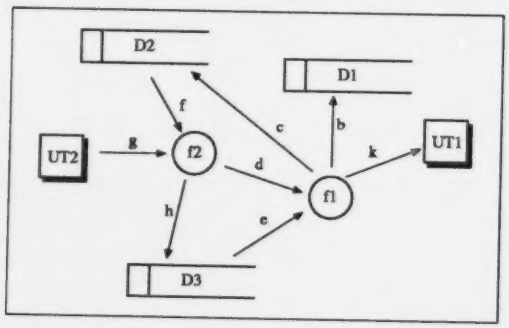


Figure 2.7: An example of ADISSA-transaction

Rules I1–I5 can be reasoned as follows. A terminator of the starting set triggers the subtask, and the control is passed from the terminator along one or several paths of CDFs. If several CDFs outgoing from a process are not AND-ed, the control is passed from the process along only one of the CDFs; otherwise the control is passed from the process along all the CDFs concurrently. Finally, the control reaches an object in the ending set which terminates the execution of the subtask.

Example 2.5 We illustrate the above rules by considering Figure 2.7. Suppose the designer has two interpretations for this ADISSA-transaction. In the first interpretation, UT2 sends data flow g to $f2$ which updates D3 according to some data stored in D2, and triggers process $f1$ which updates D1 and finally issues a report to UT1. In the second interpretation, UT1 wants a report only. To do this $f1$ will have to trigger $f2$ which reads the information stored in D2.

For the first interpretation the rules are applied as follows:

Rule (I1): UT2 belongs to the starting set.

Rule (I2): UT1 belongs to the ending set.

Rule (I3): Assume data flows d and h are linked together by AND, and data flows k, b and c are also linked by AND. So we get two expressions:

Process $f1$: k AND b AND c

Process $f2$: d AND h

Rule (I4): Data flows g, d and k are converted to CDFs. Since h is involved in an AND and involves a flow from a process, it is not converted to a CDF. The same applies to data flows b and c. Since both ANDs involve only one CDF and DFs, both ANDs are removed. Hence there is no AND in this interpretation. The resulting expressions become

Process $f1$: k

Process $f2$: d

Rule (I5): Not applicable in this interpretation.

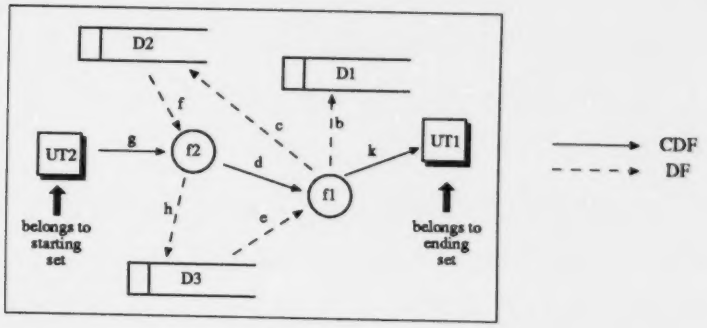


Figure 2.8: First interpretation

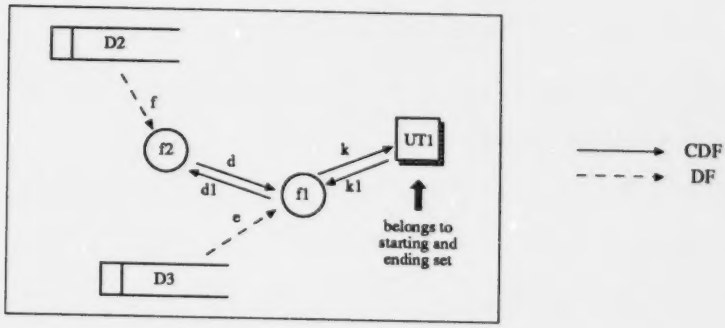


Figure 2.9: Second interpretation

The result can be seen in Figure 2.8 where a DF is shown as dashed arrow and a CDF is shown as a solid arrow.

The second interpretation of the ADISSA-transaction contains two cases where the flow of control is opposite to the direction of the original data flow: one from UT1 to $f1$ and the other from $f1$ to $f2$. The application of the five rules is presented below:

Rule (I1): UT1 belongs to the starting set.

Rule (I2): UT1 belongs to the ending set.

Rule (I3): Since data flows b, c and h are not required in this interpretation, we have the following expressions

Process $f1$: k

Process $f2$: d

Rule (I4): Only k and d are converted into CDFs.

Rule (I5): Since it contains two cases where the flow of control is opposite to the direction of the original data flow, two extra CDFs are added: $k1$ and $d1$.

The result is shown in Figure 2.9.

2.2.3.2 Merging interpretations

Rules I1–I5 specifies the process logic of each interpretation (subtask). But the process logic of the ADISSA-transaction is not yet specified, i.e., how the control is passed from the objects in the starting set to the objects in the ending set along paths of CDFs must be formally specified at this stage. This is done by applying the following rules to merge all the interpretation diagrams to form an FSM-transaction (the rules are rephrased in a more formal and organized way as opposed to the rules in [BLS91]):

(M1) Duplication of multiple-use terminators: The rule is applied separately to each interpretation diagram. The purpose of this rule is to comply with the constraints C3 and C4 (Section 2.2.3). Moreover, when a terminator plays multiple roles, e.g., in both the starting set and the ending set, each of these roles should be considered as a different state in a finite-state-machine sense.

1. If a terminator has at least one outgoing CDF and belongs to the ending set, i.e., constraint C3 is violated, one copy of the terminator is created. The duplicate has no outgoing flow and belongs to the ending set. The original terminator is removed from the ending set and is put into the beginning set instead.
2. If a terminator, say t, has n outgoing CDFs, say $d_1, \ldots, d_n$, where $n > 1$ (i.e., constraint C4 is violated), it is duplicated n times, say $t_1, \ldots, t_n$, and each t_i has only one outgoing CDF d_i. The original terminator t is removed.

The different occurrences of the terminator keep their original name, each having the same incoming CDF(s) as the original.

(M2) Merging interpretation diagrams: A single diagram is created by applying the following rules:

1. Data stores: Only one copy of every distinct data store is kept; any flow incoming to or outgoing from the data store in any interpretation is incoming to or outgoing from the merged copy.
2. Processes: Only one copy of every distinct process is kept; the flows are merged as for data stores.
3. Terminators: Terminators are not merged yet. Every occurrence of a terminator, appearing in any interpretation, is copied as is.
4. (a) If a flow is a CDF in at least one interpretation, it becomes a CDF in the resulting diagram; if not, it remains a DF.
   (b) For all groups of flows outgoing from a process, a logical expression composed of the expression of each interpretation, separated by XOR, is created. (The last statement implies only one interpretation is to be executed at a time, i.e., interpretations are not executed concurrently.)

(M3) Elimination of existing ORs in the interpretations: Since a OR b can be expressed as a XOR b XOR (a AND b), an OR-condition is not really needed.

(M4) Merging of multiple occurrences of the same terminator:

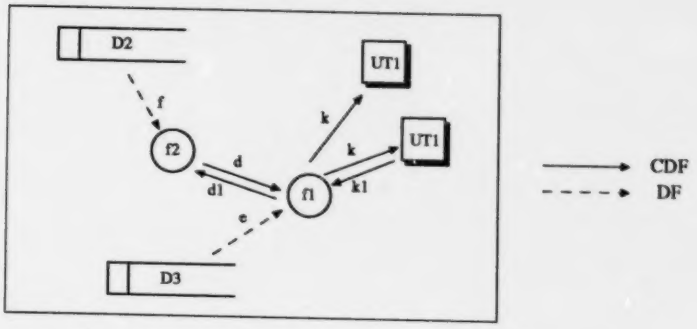


Figure 2.10: Adjustment of second interpretation

1. For a given terminator name, one copy is created for all occurrences having the same outgoing CDF; the incoming flows are connected to the copy.
2. If after that there is more than one copy of the terminator, the additional copies are given different names; the same renaming being done for the incoming flows.

After the merging process, specification is provided for the process logic of each elementary process, e.g., pseudo-code.

Example 2.6 We illustrate the rules by merging Figure 2.8 and Figure 2.9. The rules are applied as follows:

Rule (M1): UT1 in the second interpretation belongs to both the starting set and ending set. Therefore it is duplicated. Figure 2.10 shows that one occurrence of UT1 has no outgoing flow and the other has outgoing flow $k1$. Both occurrences of UT1 keep the incoming flow k.

Rule (M2): According to this rule only one copy of D1, D2, D3, $f1$ and $f2$ is kept and any flow incoming to or outgoing from each copy is copied as is. All occurrences of the terminators are still kept. All flows incoming to or outgoing from these copies of terminators are copied as they are. The expressions resulting from the merge of the two expressions are:

Process $f1$: k XOR k

Process $f2$: d XOR d

Rule (M3): This rule does not apply here.

Rule (M4): There will be two copies of UT1 and two copies of flow k. The copies will therefore be renamed.

The result is shown in Figure 2.11.

2.2.4 Converting FSM-transactions into FSMs

After we have constructed an FSM-transaction, the next step is to express the FSM-transaction as a series of FSMs. Since an FSM cannot express concurrency, we consider

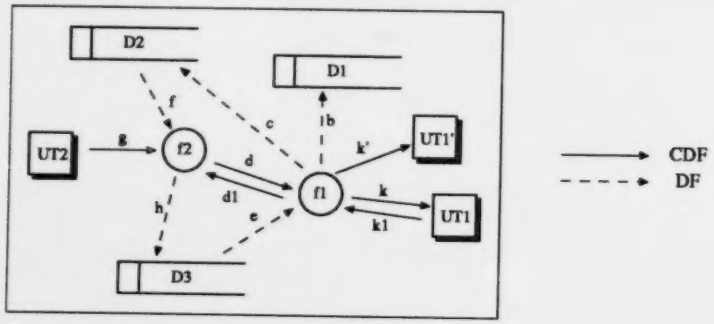


Figure 2.11: Resulting FSM-transaction

two separate cases: FSM-transactions that do not contain any CDF involved in an AND relationship, and FSM-transactions that contain CDFs involved in some AND relationships.

2.2.4.1 FSM-transactions without ANDs

For an FSM-transaction without ANDs, the control is never passed along several paths of CDFs concurrently from the objects in the starting set to the objects in the ending set. By associating object with state, and CDF with entry, we can describe the flow of control of the FSM-transaction in the finite-state-machine sense. The following steps are required [BLS91] to construct an FSM according to the definition given in Section 2.2.1:

(T1) Set S of states:

1. for each object o_i of the FSM-transaction not in the starting set, a state s_i is defined.
2. a state s_0, corresponding to all objects of the starting set, is defined.

(T2) Set E of entries:

1. for each CDF, an input entry e_i is defined.
2. an input entry e_0 which triggers the terminators of the starting set is defined.

(T3) Set A of actions: (As mentioned in Section 2.2.1, an action element is responsible for generating the next input entry to the FSM based on the current input entry and the current state.)

1. An action a_i is created for each object o_i. If o_i is a process, a_i is the execution of the associated implementation. If o_i is a terminator, a_i is the issuing of a signal on the outgoing CDF, if it has one, or the issuing of an "end-of-transaction" signal, denoted by ν (referred to as an "empty signal"). If o_i is a data store, a_i will issue the "empty signal".
2. a separate action a is added: this action involves determining which terminator of the starting set is activated, and then initiating the action corresponding to that object.

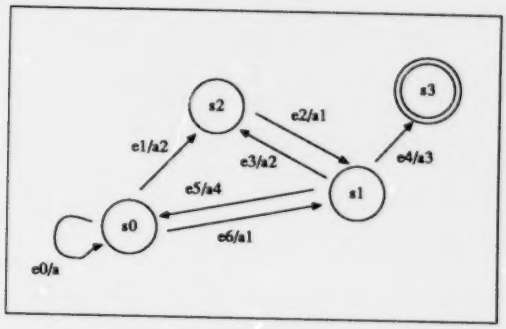


Figure 2.12: After converting to an FSM

(T4) Transformation function T:

1. If there is a CDF going from o_i to o_j in the FSM-transaction, then construct $T(s_i, e_i) = (s_j, a_j)$ where:
   - s_i is the state corresponding to o_i
   - s_j is the state corresponding to o_j
   - e_i is the entry corresponding to the CDF linking o_i and o_j
   - a_j is the corresponding action element of o_j
2. $T(s_0, e_0) = (s_0, a)$ where a is as defined above, is also part of T.

(T5) Initial state: s_0

(T6) Initial entry: e_0

(T7) Set of final states F are those states corresponding to the objects of the ending set of the FSM-transaction.

Example 2.7 We illustrate the rules by converting Figure 2.11 into an FSM. The corresponding FSM-transaction contains no ANDs. Objects and CDFs, and the corresponding states and entries are shown in Table 2.1 in which s_0 is the initial state and s_3 is the final state. Moreover, objects and the corresponding actions are shown in Table 2.2. The mapping T is shown in Table 2.3, where a is the predefined action (rule T3), and is depicted as in Figure 2.12. Note that

- when the FSM is moving to the initial state s_0 from some other states, it does not necessarily execute the predefined action a and redetermine which terminator of the starting set is activated. For example, after applying the rule T4 to the CDF k which is directed from $f1$ to UT1, we have $T(s_1, e_5) = (s_0, a_4)$. In this case, the FSM goes back to the initial state and activates the object UT1.
- the images and actions of the data stores D1, D2 and D3 are not shown in Figure 2.12, Table 2.2, Table 2.1 and Table 2.3 because we can see there are no CDFs connecting these data stores in Figure 2.11. Therefore, the states corresponding to these data stores will never be reached.

Element	Corresponding State/Entry
UT1	s_0
UT2	s_0
UT1′	s_3
$f1$	s_1
$f2$	s_2
g	e_1
d	e_2
$d1$	e_3
k'	e_4
k	e_5
$k1$	e_6

Table 2.1: Elements and corresponding states/entries

Object	Corresponding Action
UT1	a_4
UT2	a_5
UT1′	a_3
$f1$	a_1
$f2$	a_2

Table 2.2: Objects and corresponding actions

	e_0	e_1	e_2	e_3	e_4	e_5	e_6
s_0	(s_0, a)	(s_2, a_2)					(s_1, a_1)
s_1				(s_2, a_2)	(s_3, a_3)	(s_0, a_4)	
s_2			(s_1, a_1)				
s_3							

Table 2.3: Mapping T

2.2.4.2 FSM-transactions with ANDs

For an FSM-transaction that contains ANDs, we decompose it into several smaller FSM-transactions, each of which does not contain ANDs. The idea is as follows. When several CDFs outgoing from a process are AND-ed, we add a new process, *calling process*, and the control is passed from the process to the *calling process* instead of to all the CDFs concurrently. Each of these CDFs is placed in a new separate FSM-transaction. The *calling process* will initiate these new FSM-transactions concurrently, and wait until all of them terminate. Finally the *calling process* terminates the FSM-transaction in which it is contained.

We apply the following rules to decompose an FSM-transaction into several smaller FSM-transactions (the rules are rephrased from [BLS91]):

(S1) First a *transaction* graph from the FSM-transaction is created. It is defined as a directed graph where each node is associated with an object of the FSM-transaction, and each edge is associated with a CDF. This graph is named G_0. (As described earlier in Section 2.2.3, elementary processes, terminators and data stores are referred as objects.)

(S2) If there are n CDFs $d_1, \ldots, d_n$ that are involved in ANDs, we duplicate the graph G_0 n times and label them G_1 to G_n.

(S3) We define the *entry nodes* of G_0 to be all the objects in the starting set. For G_i, $1 \leq i \leq n$, we define the *entry edge* of G_i to be d_i.

(S4) For each G_i, we visit the directed graph starting from the corresponding entry nodes/edge in all the possible paths. We mark every node and edge that we can reach. (The idea is to trace the paths of flow of control.) It is possible that not every node and edge can be reached. They will be deleted in the next rule.

(S5) For each G_i, delete all the nodes and edges that are not marked. (If the control cannot reach certain objects, those objects will not be considered in the construction of the FSM-transactions.)

(S6) Every edge v_i of G_k which is the entry edge to any G_j, where $j \neq k$, is removed from G_k. (If v_i is the entry edge of G_j, it implies v_i is involved in ANDs. As mentioned at the beginning of this section, v_i is to be placed in a separate FSM-transaction and therefore should not belong to G_k.)

(S7) Clear the mark of every G_i.

(S8) For each G_i, we revisit the graph starting from the entry nodes/edge. Mark every node and edge that we reach.

(S9) For each G_i, we delete all the nodes and edges that are not marked.

(S10) Replace each node by the original object, and each edge by the original CDF.

Each G_i will not contain an AND. In the next stage, we add some *administrative objects* to each G_i to make it satisfy the constraints of FSM-transaction. There are four kinds of administrative objects:

1. *Entry terminator*: It transmits the flow from the calling to the called transactions. It is applied to G_k for $1 \leq k \leq n$.

2. *Transmission process*: It transmits the flow it receives. It is introduced to avoid the case of two terminators connected directly.

3. *Calling process*: As discussed at the beginning of this section, a *calling process* executes an AND. It activates the transactions by transferring the control to the transactions involved. It then waits for the ending of all transactions; and finally, it issues a signal to the corresponding exit terminator.

4. *Exit terminator*: The object signifies the end of all subtransactions activated by an AND.

For every G_i $(0 \leq i \leq n)$, we do the following:

(R1) If $G_i \neq G_0$, an entry terminator, a transmission process, and a CDF from the entry terminator to the transmission process are created. In addition, the initial endpoint of the entry edge d_i is changed to be the transmission process, and the terminal endpoint of the d_i remains unchanged.

(R2) If a process of G_i has outgoing CDFs linked by AND, then for each group of ANDs, the following elements are introduced:

1. a calling process
2. a CDF linking the process to the calling process.
3. an exit terminator
4. a CDF linking the calling process to the exit terminator

Now each G_i is an FSM-transaction without ANDs and can be transformed into an FSM.

Example 2.8 We illustrate the rules by considering an FSM-transaction as shown in Figure 2.13. Suppose the logical expression associated with $f1$ is

$$(a \text{ AND } b) \text{ XOR } (a \text{ AND } c) \text{ XOR } (b \text{ AND } c).$$

First we create the graph G_0 (rule S1) which is shown in Figure 2.14. Since the flow g is not a CDF, it does not have any corresponding edge in Figure 2.14. There are three data flows that are involved in ANDs. Hence G_0 is duplicated three more times, named G_1, G_2 and G_3 respectively (rule S2). We set the entry nodes/edges for $G_0, \ldots, G_3$ to be UT1, a, b and c respectively (rule S3). We apply the rule S4 and use the symbol $\surd$ to mark each node or edge that we reach. The result is shown in Figure 2.15. Next we apply the rules S5 and S6 to delete all nodes and edges that are not marked or are the entry edges to other graphs. The result is shown in Figure 2.16. G_1– G_3 remain unchanged after applying the rules S7–S9. Figure 2.17 shows the result of G_0 after applying the rules S7–S9. Now we apply the rule S10 and get four small incomplete FSM-transactions as shown in Figure 2.18. Now we expand each of these graphs by adding administrative objects. For G_0, there are three groups of ANDs and we apply the rule R2 and add three different "calling processes"

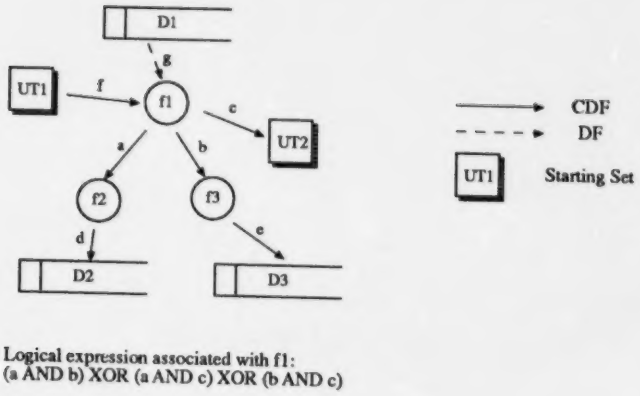


Figure 2.13: An FSM-transaction contains ANDs

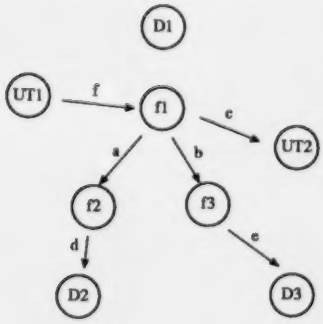
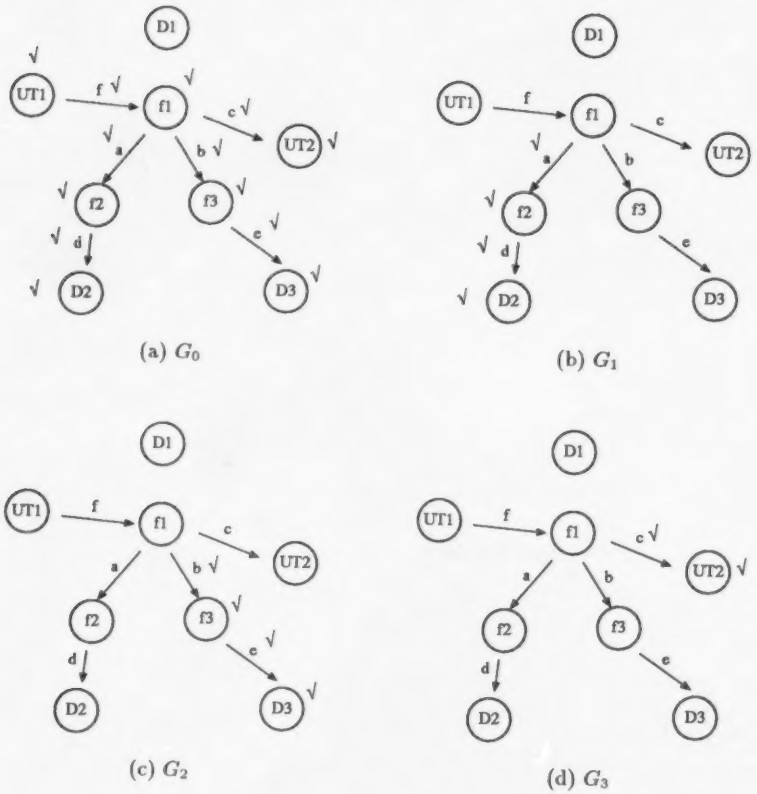


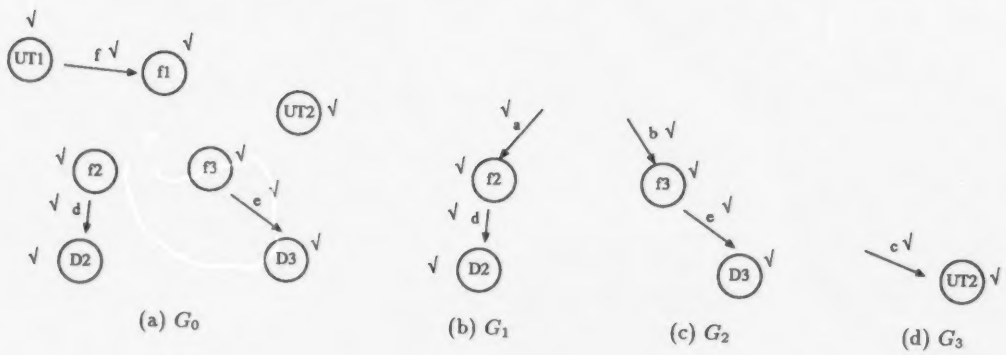
Figure 2.14: Transaction graph G_0



(a) G_0 (b) G_1

(c) G_2 (d) G_3

Figure 2.15: After applying the rule S4



(a) G_0 (b) G_1 (c) G_2 (d) G_3

Figure 2.16: After applying the rules S5 and S6

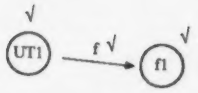
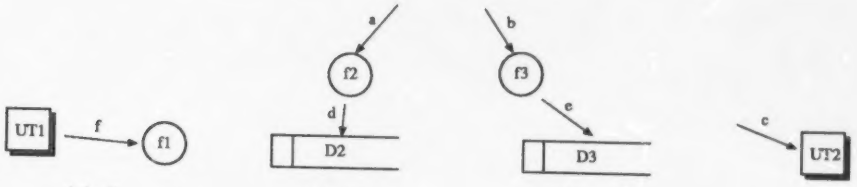


Figure 2.17: After applying the rules S7–S9 to G_0



(a) G_0 (b) G_1 (c) G_2 (d) G_3

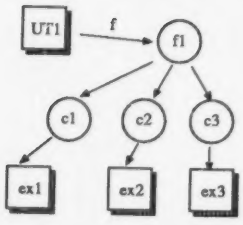
Figure 2.18: After applying the rule S10

(c_1, c_2 and c_3) and three different "exit terminators" (ex_1, ex_2 and ex_3). This is shown in Figure 2.19(a). For G_1 to G_3, we apply the rule R1 and expand the graph by adding "entry terminators" (en_1, en_2 and en_3) and "transmission processes" (t_1, t_2 and t_3). This is shown in Figure 2.19(b)–(c). Now each FSM-transaction does not contain any ANDs. So we can apply the rules T1–T7 to convert each of the FSM-transaction into an FSM. The diagrams of the final FSMs are omitted because the resulting FSMs will look almost the same as the Figures 2.19 and Example 2.7 has provided an example of applying the rules T1–T7.

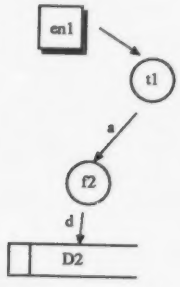
2.3 Summary

In this chapter we have studied the ADISSA methodology and its related transformation. ADISSA is a transaction-oriented refinement of Structured Analysis. Hierarchical data flow diagrams are used in the analysis stage. From these DFDs, ADISSA-transactions are identified by partitioning the set of all elementary processes. Interpretations (subtasks) of each ADISSA-transactions are identified by specifying the starting set, the ending set, control data flows, and logical expressions of the outgoing data flows of processes. Finally the process logic of an ADISSA-transaction is formalized by merging all the interpretations to become an FSM-transaction. Finite state machines are used as design primitives in the design stage, and they are constructed from the FSM-transactions produced in the analysis stage by applying a series of formal transformation rules. They provide a formal description of the FSM-transactions in terms of the flow of control.

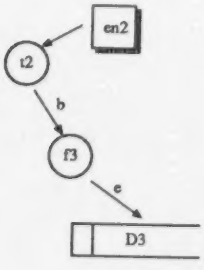
The notions of ADISSA-transactions, interpretations, and FSM-transactions are used in subsequent chapters. Chapter 3 discusses how metrics can be applied to improve the transformation process. Chapter 5 discusses how ETL is used to represent these intermediate structures. All the transformation rules required to build these intermediate structures are formally and rigorously presented in this chapter, and they will be defined using ETL in Chapter 6.



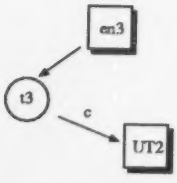
(a) Transaction resulting from G_0



(b) Transaction resulting from G_1



(c) Transaction resulting from G_2



(d) Transaction resulting from G_3

Figure 2.19: Resulting transactions after applying the rules R1 and R2

Chapter 3

Use of Metrics on ADISSA Methodology

A typical software life cycle consists of requirements analysis, design and implementation phases which are shown in Figure 3.1. The use of such a life cycle involves implicit evaluations and iteration. For example, decisions made during the design phase may necessitate a revision of the requirements analysis specifications before returning to design. This type of iteration is indicated by the arrows pointing upward in Figure 3.1.

As mentioned in Chapter 1, it is desirable to introduce software metrics *explicitly* to each phase of the life cycle to

1. control the size and complexity of the products produced in each phase by giving immediate feedback to the designer so that corrective actions can be taken earlier;
2. help designers in decision making; and
3. guide transformations from phase to phase.

In this chapter we study and analyze some metrics which can be applied to the ADISSA methodology and its related ADISSA transformation. Software metrics will be applied before, during and after the transformations. Specially, metrics will be applied to the hierarchical DFD, ADISSA-transactions and FSMs. The software metrics we study are not all the metrics that can be applied to the ADISSA methodology and its related transformation method. The main concern is to show the feasibility of coexistence of software metrics with software transformation methods. Section 3.1 presents some metrics which can be applied

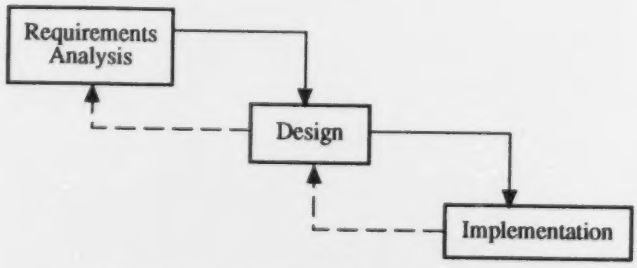


Figure 3.1: A Traditional Life Cycle

to DFDs in general. Section 3.2 discusses the use of SEC metrics. Its use is related to the control of the size of ADISSA-transactions. Section 3.3 introduces metrics that can be applied to an FSM-transaction. Section 3.4 describes the use of McCabe's cyclomatic complexity to an FSM. The final section summarizes the chapter.

3.1 Metrics in the DFD Environment

Based on DeMarco's count of components metrics [DeM82], the following metrics can be applied to a DFD:

PROC: Number of elementary processes in a DFD

DEI: Total number of data elements moving into an elementary process

DEO: Total number of data elements moving out of an elementary process

DFI: Total number of data flows incoming into an elementary process

DFO: Total number of data flows outgoing from an elementary process process

During DFD specification, two common problems are

1. to decide the number of process in a data flow diagram, and
2. to decide the maximum level of decomposing processes.

For the first problem, it is recommended the DFD should be fit on one page [DeM78] or one screen [MM85]. The metric PROC will assist the designer in producing understandable and uniform designs by limiting the number of processes that should be contained in a data flow digram. When the value returned by PROC is high, the designer should look for a cluster of elementary processes that can be replaced by a general process (i.e., bottom up approach to requirements specification). For the second problem, knowing the number of data flows or data elements around the boundary of the process (by computing the metrics DEI, DEO, DFI and DFO), it is possible to decide if it is advisable to reorganize processes or to decompose processes further. For example, the elementary process p in Figure 3.2(a) has several data flows entering and going out. If the specification of p is known in advance, or can be refined by additional analysis, we can replace p by a cluster of processes as shown in Figure 3.2(b) where the number of data flows/elements around the boundary of each process is less than p. This can be determined by using DEI, DEO, DFI, and DFO. If replacing p by Figure 3.2(b) will cause the number of elementary processes in the diagram to become greater than a predetermined maximum, the designer is advised to make p non-elementary and let p explode into Figure 3.2(b). In this case, the designer should also examine the data elements associated with the data flows and allow some aggregation of the data flows.

3.2 Controlling the Complexity of ADISSA-Transactions

Each ADISSA-transaction is constructed by partitioning the set of all elementary processes into components. Each component is modified and incorporated with other elements (data stores, data flows, terminators) to become an ADISSA-transaction. Two elementary processes are in the same component and hence in the same ADISSA-transaction if they are

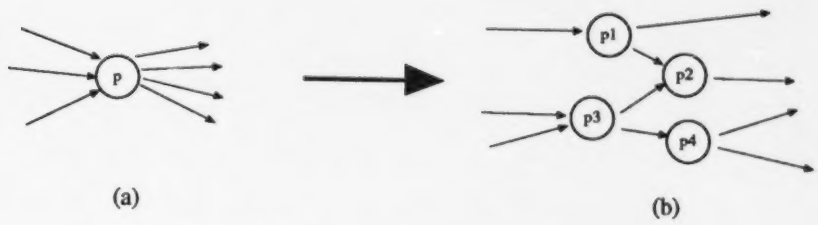


Figure 3.2: Process that needs refinement

Figure 3.3: Before adding a data store between p and q

connected by a data flow. Since an ADISSA-transaction is a "flat" DFD (i.e., explosions of processes are ignored), any metrics that can be applied to a flat DFD can be applied to the ADISSA-transaction as well. Besides these metrics, we can devise the following metrics for an ADISSA-transaction:

SEC: Components size resulting from single edge cut

When the number of elementary processes in an ADISSA-transaction is large, it is possible to split the ADISSA-transaction into two smaller ADISSA-transactions. Suppose p and q are two elementary processes and there is a data flow d from p to q as shown in Figure 3.3. Processes p and q will then be in the same ADISSA-transaction, say A. If we replace d by a data store as shown in Figure 3.4, then p and q will be forced into different ADISSA-transactions, unless p and q are contained in a cycle (i.e., there is another path which consists of elementary processes joining p and q). Figure 3.5 depicts an example where p and q cannot be put in different ADISSA-transactions if d is to be replaced by a data store. Assume after introducing a new data store, p and q are contained in the ADISSA-transactions, say A_1 and A_2 respectively. The metric SEC will take d as argument and return the 3-tuple (k, m, n) where k is the number elementary processes contained in A, m is the number of elementary processes contained in A_1, and n is the number of elementary processes contained in A_2. If $k = m = n$, that means there is a cycle containing p and q in the original ADISSA-transaction A.

The metric SEC is objective in the sense it is well-defined that two people should compute the identical value from the same ADISSA-transaction independently. Moreover, it is not costly in the sense that the computation can be performed easily. The metric SEC can help the designer to

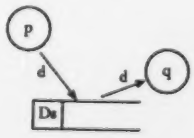


Figure 3.4: After adding a data store between p and q

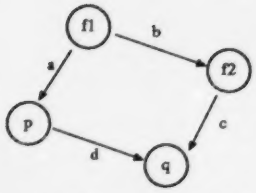


Figure 3.5: p and q are contained in a cycle

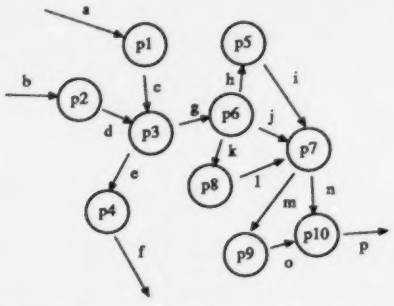


Figure 3.6: An example of ADISSA-transaction

1. select the most appropriate data flow to be replaced by a data store by considering the sizes of resulting smaller ADISSA-transactions;

2. detect cycles in the DFD.

When the SEC metric suggests an ADISSA-transaction to be refined, the designer should go back to the initial hierarchical DFDs and consider where the new data store should be added these DFDs, and restarts the transformation.

Example 3.1 Figure 3.6 shows an example of an ADISSA-transaction. For convenience, the terminators and the data stores are not drawn. We apply the metric SEC to each data flow that connects two elementary processes. The result is shown in Table 3.1. A good break point would be to insert a data store between the processes p_3 and p_6 cutting data flow g. For data flow h, there is no way to place processes p_5 and p_6 in different ADISSA-transactions because they are part of the same cycle.

The metric SEC considers only one data flow at a time. A natural extension will be to consider more than one data flow at a time, e.g. two data flows at a time, three data flows at a time, and so on. Since there is no termination to this extension and a metric should not be too complicated to compute, we restrict to consider one data flow at a time.

3.3 Controlling the Complexity of FSM-Transactions

The size and complexity of an FSM-transaction cannot be controlled independently without taking the corresponding ADISSA-transaction into consideration. All the components of an FSM-transaction, except the introduction of multiple occurrences of terminators, control data flows (CDFs) and logical expressions, come from the corresponding ADISSA-transaction. No matter how many interpretations of an ADISSA-transaction is to contain, a

Data flow	SEC metrics
c	$(10, 1, 9)$
d	$(10, 1, 9)$
e	$(10, 9, 1)$
g	$(10, 4, 6)$
h	$(10, 10, 10)$
i	$(10, 10, 10)$
j	$(10, 10, 10)$
k	$(10, 10, 10)$
l	$(10, 10, 10)$
m	$(10, 10, 10)$
n	$(10, 10, 10)$
o	$(10, 10, 10)$

Table 3.1: Results of SEC metrics

new component will not be added to the FSM-transaction after merging the interpretations. If the ADISSA-transaction is already complex, the complexity will be carried over to the FSM-transaction.

If the logical expressions introduced do not contain any AND-relationships, they will not increase the complexity of the FSM-transaction. If the logical expressions contain some AND-relationships, a series of FSMs will be constructed by the rules S1–S10 and R1–R2 (Section 2.2.4.2). As a result, the FSM-transaction is replaced by a series of smaller FSM-transactions each of which is less complex than the original FSM-transaction.

At this stage, it would be better to proceed with the transformation and control the size and complexity of the target FSMs. If an area of high complexity is detected in the target FSMs, the designer can trace it back to the ADISSA-transactions and take corrective actions.

3.4 Metrics in the FSM Environment

The following metrics can be applied to an FSM:

ST: The number of states in an FSM

STI: The number of state transitions into a state

STO: The number of state transitions out of a state

These metrics are similar to the PROC, DFI, and DFO metrics mentioned in Section 3.1.

Besides the above metrics, a variation of McCabe's cyclomatic complexity [McC76] can be applied to an FSM. By applying the theory of cyclomatic number in graph theory [Ber73] to a control flow graph of a program, McCabe develops a complexity measure of a program:

$$\nu(G) = e - n + 2 \tag{3.1}$$

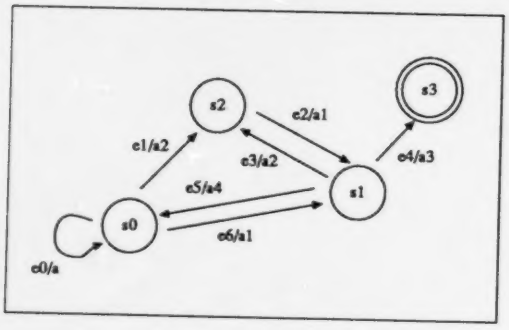


Figure 3.7: An FSM

where G is the control flow graph of the program, e is the number of edges, and n is the number of nodes in G. A control graph represents a procedural unit of a program in which there is a unique entry and a unique exit point. The nodes denote sequential blocks of code and the edges mirror decisions causing a branch. It is assumed that each node can be reached from the entry node and each node can reach the exit node. If the graph G is planar, then $\nu(G)$ is equal to the number of regions in the graph G. The cyclomatic complexity $\nu(G)$ is designed to indicate a procedure's testability and maintainability by calculating the maximum number of *linearly independent*[1] paths through the program [McC76]. By associating state with node, and state transition with edge, we can, in a similar fashion, compute the cyclomatic number of an FSM.

The FSM defined by Babin *et al.* (Section 2.2.1) has one initial state and can have more than one final state. In order to apply Equation (3.1), we add a pseudo-final state to the FSM. If there are f final states in the FSM, say $s_1, \ldots, s_f$, we introduce f new input entries, say $e_1, \ldots, e_f$, to the FSM so that for $1 \leq i \leq f$, whenever the FSM is at state s_i and the current input entry is e_i, the FSM will move to the pseudo-final state and stop. Then the FSM has a unique entry and exit point and has the same functionality as the original one. If the original FSM has n states, and e state transitions, then in the new FSM there are $n+1$ states and $e+f$ transitions. The cyclomatic complexity of the FSM becomes

$$\begin{aligned} & (e+f)-(n+1)+2 \\ = \; & e+f-n+1 \end{aligned} \tag{3.2}$$

When there is only one final state, i.e., $f = 1$, Equation (3.2) is the same as Equation (3.1). When the cyclomatic complexity of an FSM is too high, it indicates that there is a state from which the FSM can move to several different states.

Example 3.2 Consider Figure 3.7 which is copied from Figure 2.12. Since the graph is planar and contains 5 regions, the cyclomatic complexity is 5. The same result is obtained by counting the number of nodes, edges and final states. The FSM contains 4 nodes, 7 edges and 1 final state. Therefore, the cyclomatic complexity of the FSM is $7+1-4+1=5$ which agrees with the previous computation.

[1] A vector space can be associated with the set of all cycles in a graph G. The cyclomatic number of a graph G is equal to the dimension of the associated vector space [Ber73]. Hence the phrase *linearly independent* is used in a vector-space sense.

3.5 Summary

In this chapter, we have studied some metrics that can assist the ADISSA transformation. The metrics can be applied before, during, and after the transformation. Metrics applied to a DFD are PROC, DEI, DEO, DFI and DFO. PROC computes the number of elementary process in a DFD. DEI, DEO, DFI and DFO compute the number of data elements and data flows around the boundary of a process. A metric, SEC, is devised to count the components size resulting from a single edge cut. Its use is related to the control of the size of an ADISSA-transaction. Metrics applied to a FSM are ST, STI and STO. ST computes the number of states in an FSM. STI and STO compute the number of state transitions into and out of a state. Another metric, a variation of McCabe's Cyclomatic Complexity, is devised to determine the cyclomatic number of an FSM. The metrics we have studied are not all the metrics that can be applied to the ADISSA methodology. The main concern is to show the feasibility of coexistence of software metrics with software transformation methods.

In the next chapter, the EARA model of Metaview will be introduced. Moreover, the DFD and the FSM environments used in the ADISSA methodology will be formalized in Metaview using the environment definition language (EDL).

Chapter 4

Metaview for Many Specification Environments

In this chapter we study the EARA (Entity-Attribute-Relationship-Aggregate) model on which Metaview [STM88, McA88] is based. In Section 4.1, the EARA model is described as a simple, yet expressive model that is sufficient for representing many modern CASE environments. In Section 4.2, the definition of an EARA database state is described in preparation for the definition of *transformation aggregate* which will be described in Chapter 5. In Section 4.3, DFD and FSM environments used in ADISSA methodology are modeled in EARA. The final section summarizes the chapter.

4.1 EARA Model

The EARA model is an enhancement of Chen's entity-relationship model [Che76]. It provides explicit representation of entities and relationships. *Entities* represent concepts or things in a specification such as processes or data. A *relationship* can exist between two or more entities. As an example, in a Data Flow Diagram environment, a process entity may send a data element entity to another process entity. This is modeled as a ternary relationship called 'sends' with the three entities as participants (process, data element, process).

The model also allows for a collection of entities and relationships to be represented by a single database object called an *aggregate*. An aggregate provides a context for defining some part of the specification. An example aggregate would be a process explosion in a data flow environment.

Attributes can be associated with entities, relationships, and aggregates. Intrinsic attributes are provided for the name and synonym of entities and aggregates as well as the type, the creator's name, the creation time, and last modification time of all EARA objects. A number of built-in value types are provided, including integers, real numbers, fixed length strings, and boolean values. Other types include identifiers for naming objects, date-time for time stamps, text for description of varying length, records for representing cartesian products of values, and enumerated types which define a finite set of name values.

The EARA model supports inheritance through generalization hierarchies. For example, the process type in a Data Flow Diagram environment may be generalized into a single universal object entity type.

The EARA model contains a number of sets and functions for defining an environment model. Each environment definition is described using the following sets and functions:

AgT – a finite, non-empty set of **aggregate** types.
ET – a finite, non-empty set of **entity** types.
RT – a finite set of **relationship** types.
AN – a finite set of **attribute** names.
RN – a finite set of **role** names.
VT – a finite set of **value** types.
$ct : AgT \rightarrow \text{powerset}(ET \cup RT)$
– the **component type** function.
$ai : ET \rightarrow AgT$
– the **aggregate identification** function (partial).
$aa : (AgT \cup ET \cup RT) \rightarrow \text{powerset}(AN \times VT)$
– the **attribute association** function.
$rm : RT \rightarrow \text{powerset}_2(RN \times \{\text{single, list}\})$
– the **role mapping** function.
$rpt : RT \rightarrow \text{powerset}_1(\text{powerlist}_2(ET \cup AgT))$
– the **relationship participant type** function.
$ag : AgT \rightarrow AgT$
– the **aggregate generalization** function (partial).
$eg : ET \rightarrow ET$
– the **entity generalization** function (partial).
$rg : RT \rightarrow RT$
– the **relationship generalization** function (partial).

where

- powerset(A) denotes the set of finite subsets of A,
- $\text{powerset}_k(A)$ is the set of all finite subsets of A which contain at least k elements,
- powerlist(A) is the set of all finite lists formed from elements of A, and
- $\text{powerlist}_k(A)$ is the set of all finite lists formed from elements of A and containing at least k items.

The elements of AgT, ET, RT, RN and AN are identifiers. To promote clarity, these sets are pairwise disjoint. The names of the intrinsic attributes are members of the set AN for all environments. The set VT contains the value types allowable for the attributes in a given environment. The basic types are members of this set for all environments.

The functions define relationships between the elements of the sets. The *component type* function (ct) defines what types of entities and relationships may be components of aggregates of a given type. Each aggregate type must have at least one component type.

The *aggregate identification* function (ai) maps an entity type name to an aggregate type name. This allows an aggregate which is composed of many entities and relationships, to be treated as a single identifiable entity within another aggregate.

The *attribute association* function (aa) function defines the set of attributes that are defined for objects of a given aggregate, entity, or relationship type. This function also

defines the value type for each attribute. Attributes for different object types may use the same name but an attribute name for a given type may not be defined more than once. Also since the intrinsic attributes are defined for all environments, the mappings of the attribute association function for these attributes are implicitly defined.

The *role mapping* (rm) and *relationship participant types* (rpt) functions define the participants of a relationship. The role mapping function defines the set of role names associated with a given relationship type. As with attributes, a given role name may be used by a number of relationship types but must not be used more than once for a given relationship type. Also defined with each role name is either of the identifiers *single* or *list* which defines the cardinality of the participant for the role as either one or many, respectively. The relationship participant types function specifies the entity and/or aggregate types whose object may participate in each role. Multiple combination of participant types for a relationship type may be defined.

The generalization functions (ag, eg, rg) map object subtypes to their appropriate supertypes for aggregates, entities and relationships. These functions are partial functions since not all types are subtypes of other types. A subtype inherits the attributes from the supertype, thus, names in the subtype must not conflict with names in the supertype. For aggregates, the subtypes also inherit the component types of the supertype. A component type function mapping from an aggregate type to an entity or relationship holds if there exists such a mapping from one of the aggregate's supertypes.

To define an environment based on the EARA model, the environment definition language EDL has been created. The relationship between the language and the EARA model constructs are generally very obvious, and a full definition of EDL can be found in [McA88]

4.2 EARA Database State

Once a particular environment has been defined, specifications can be stored based on that definition. A *database state* is a stored representation of a given specification.

The database state is characterized by the following sets and functions:

AgI – a finite set of aggregate instances.
EI – a finite set of entity instances.
RI – a finite set of relationship instances.
V – a finite set of values.
$agt : AgI \rightarrow AgT \backslash ag(AgT)$
– the **aggregate type** function where
\ is the set difference operator.
$et : EI \rightarrow ET \backslash eg(ET)$
– the **entity type** function.
$rt : RI \rightarrow RT \backslash rg(RT)$
– the **relationship type** function.
$vt : V \rightarrow VT$
– the **value type** function.
$vm : ((AgI \cup EI \cup RI) \times AN) \rightarrow V$
– the **value mapping** function (partial).
$rpi : (RI \times RN) \rightarrow \text{powerlist}_1(EI \cup AgI)$
– the **relationship participant instance** function.

$cm : (EI \cup RI) \rightarrow AgI$
– the **component mapping** function.

The elements of AgI, EI and RI are instances that make up an EARA database state. The aggregate type function (agt) requires the type of every aggregate instance to be chosen from the set AgT. Moreover no supertypes can have any instance. Similar rules apply to the functions et and rt. The value type function (vt) restricts the values to be predefined by the value type set VT.

The value mapping function (vm) assigns values to attributes that are associated with aggregates, entities and relationships instances.

The relationship participant instance function (rpi) defines the participant instances of every relationship instance.

Each aggregate instance is composed of a number of entities and relationships instances, each of which is a component of *one and only one* aggregate instance. The component mapping function (cm) establishes this correspondence between aggregates instances and their component entities and relationships instances.

The concept of an aggregate is included as part of the EARA model so that a specification database state can be partitioned into a collection of relatively independent information modules. Each aggregate describes a certain part of a software system and an analyst should be able to work in the context of a given aggregate without having to be concerned with the rest of the system. Thus each instance in a specification database state is identified by the following criteria:

- A given aggregate instance is uniquely identified by its name.
- A given entity instance is uniquely identified by its name and the name of the aggregate of which it is a component.
- A given relationship instance is identified by its type, its list of participants and the name of the aggregate of which it is a component.

Besides the above identification criteria, the following rule applies to entities that make up a specification database state [McA88]:

- Let E be an entity instance that is a component of the aggregate A. E may participate only in those relationships that are components of A. In other words, there is no relationship instance that can have entity participant instances come from different aggregates.

Rules and conditions that apply to the construction and consistency of a database state can be found in [McA88].

4.3 Modeling the DFD and FSM Environments

In this section, DFD and FSM specification environments are represented using the Environment Definition Language (EDL). The ETL specification of the ADISSA transformation (presented in Chapter 6) relies on the EDL specifications of both environments to perform the transformation process.

4.3.1 Modeling the DFD Environment

The EDL specification for the DFD environment is (comments are parenthesized with /* ... */):

```
/* Entities */
 ENTITY_TYPE universal  GENERIC
      ATTRIBUTES (description: text);
 ENTITY_TYPE dataObject  GENERIC  IS_A universal
                  tree  IS_A universal
                  /* building block for logical expression */
                  process  IS_A universal
                  dataFlow  IS_A dataObject
                  dataElement  IS_A dataObject;
 ENTITY_TYPE terminator  IS_A universal
/* There are four different kinds of terminators. */
      ATTRIBUTES (type: (user, time, real_time, communication));
 ENTITY_TYPE dataStore  IS_A universal
      ATTRIBUTES (form: string(1...30));

/* Relationships */
 RELATIONSHIP_TYPE sends
/* describes data flows connecting objects */
      ROLES (source, data, destination)
      PARTICIPANTS (process, dataFlow, process | processDecomposition | terminator)
                   (terminator | processDecomposition, dataFlow, process)
                   (processDecomposition, dataFlow | dataFlowGroup,
                   processDecomposition | terminator)
                   (terminator, dataFlow | dataFlowGroup, processDecomposition);
 RELATIONSHIP_TYPE access  GENERIC
      ROLES (name, flow, data)
      PARTICIPANTS (process, dataFlow, dataStore);
 RELATIONSHIP_TYPE changes  IS_A access;
/* write, update, delete to data stores */
 RELATIONSHIP_TYPE reads  IS_A access;
/* retrieve data from data stores */
 RELATIONSHIP_TYPE dataElementsInFlow
/* associates data to data flows */
      ROLES (flow, data)
      PARTICIPANTS (dataFlow, dataElement);
 RELATIONSHIP_TYPE dataStored
      ROLES (store, data)
      PARTICIPANTS (dataStore, dataElement);
```

```
RELATIONSHIP_TYPE connective
/* logical expressions */
    ROLES (root, left_tree, right_tree)
    PARTICIPANTS (tree, tree | dataFlow, tree | dataFlow)
    ATTRIBUTES (type: (and_type, xor_type));
RELATIONSHIP_TYPE associated_tree
/* associate logical expression to process */
    ROLES (pname, top_root)
    PARTICIPANTS (process, tree);

/* Aggregates */
AGGREGATE_TYPE contextDiagram
/* Root data flow diagram */
    COMPONENTS (terminator, processDecomposition, dataFlowGroup, sends)
    ATTRIBUTES (description: text);
ENTITY_TYPE processDecomposition
    BECOMES processExplosion;
AGGREGATE_TYPE processExplosion
/* allow a process to be exploded into a more detailed DFD */
    COMPONENTS (process, dataFlow, dataFlowGroup, dataStore, tree
                processDecomposition, sends, access, connective, associated_tree)
    ATTRIBUTES (description: text);
ENTITY_TYPE dataFlowGroup
    BECOMES dataFlowPack;
AGGREGATE_TYPE dataFlowPack
/* allow several data flows to be treated as a single data flow */
    COMPONENTS (dataFlow);
    ATTRIBUTES (description: text);
```

The DFD environment used in the ADISSA methodology differs from a DFD environment (e.g., [DeM78]) in that

- Terminators are further divided into four different kinds: *user terminator*, *time terminator*, *real-time terminator* and *communication terminator*.
- Logical expression can be associated to outgoing data flows of a process.

To capture such a difference, we add a new attribute to the entity type *terminator* and a new entity type *tree* and two new relationship types *connective* and *associated_tree*.

In the EDL specification of DFD environment, terminators are treated as entities and have an attribute *type* which can only take values from the enumerated set of identifiers: { user, time, real_time, communication }.

Since operator OR can be expressed in terms of AND and XOR, we eliminate it in our EDL specification. Entities of type *tree* are building blocks for building logical expressions. An instance of type *tree* is used to label a binary tree which is an instance of the relationship type *connective* where the left child and the right child can be other binary trees or data flows. Since AND and XOR are binary operators, every logical expression that is constructed

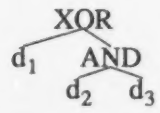


Figure 4.1: Graphical Presentation of Logical Expression

from AND and XOR can be represented graphically as a binary tree. For example, for the logical expression

$$d_1 \quad \text{XOR} \quad (d_2 \quad \text{AND} \quad d_3),$$

we can present it graphically as in Figure 4.1.

SSL (Simple Specification Language) is a language of Metaview which can be used to describe an EARA database state [McA88]. The SSL specification of the above logical expression is:

```
dataFlow d1, d2, d3;
tree a, b;
connective (b, d2, d3) (type := and_type);
connective (a, d1, b) (type := xor_type);
```

Relationship type *associated_tree* links a logical expression to a process. The rule here is to bind the top root of a binary tree to a process. For example, if p is an instance of type *process* and has a logical expression specified in SSL as above, we bind the logical expression to p via the following SSL statement:

```
associated_tree(p,a);
```

Control data flow (CDF) is not part of the EDL specification because it only exist during transformation.

4.3.2 Modeling the Finite State Machine Environment

The EDL specification for the Finite State Machine environment is:

```
/* Entities */
ENTITY_TYPE universal GENERIC
    ATTRIBUTES (description: text);
ENTITY_TYPE state IS_A universal
    ATTRIBUTES (type_of_state: (initial, final, intermediate),
                administrative_object: boolean);
ENTITY_TYPE entry IS_A universal
    ATTRIBUTES (type_of_entry: (initial, empty, intermediate));
ENTITY_TYPE action IS_A universal;

/* Relationships */
RELATIONSHIP_TYPE has_subtransactions
    ROLES (calling_function, subtransaction: LIST)
    PARTICIPANTS (state, fsm_aggregate);
RELATIONSHIP_TYPE next_state_function
    ROLES (input_state, input_entry, output_state, action_element)
    PARTICIPANTS (state, entry, state, action);
```

```
/* Aggregates */
 AGGREGATE_TYPE fsm_aggregate
     COMPONENTS (ALL)
     ATTRIBUTES (description: text);
```

A finite state machine environment consists of a number of states, actions, and input entries plus a next-state function. States, actions and input entries are treated as entities with types *state*, *action*, and *entry* respectively in our EDL specification. Moreover a state is further characterized by specifying whether it is an initial state, a final state or an intermediate state. An extra boolean attribute *administrative_object* is added to indicate whether it is the image of a process in the source DFD environment during the ADISSA transformation. Rules R1 and R2 of the ADISSA transformation (Section 2.2.4.2) will create some new administrative objects (calling processes, transmission processes, etc.) which do not exist in the source DFD environment. An input entry is characterized by classifying whether it is an initial entry, empty entry or intermediate. It is done by adding an attribute *type_of_entry*.

Relationship type *has_subtransactions* is used to link the administrative object *calling process* to the other FSMs to which the control is being transferred.

As mentioned in Section 2.2.1, there is a mapping $T: S \times E \rightarrow S \times A$ in the definition of FSM proposed by Babin *et al.* where S is the set of states, E is the set of input entries and A is the set of actions. During the ADISSA transformation S becomes the set of all terminators, data stores and processes, E becomes the set of CDFs and A becomes the set of execution of elements in S (Section 2.2.4.1). This mapping T is modeled by the relationship type *next_state_function*.

4.4 Summary

In this chapter, we have described the EARA model and the EARA database state. The EARA model is a simple, yet expressive model that is sufficient for representing many modern CASE environments (e.g., DFD and FSM specification environments are represented using the EDL in this chapter). An EARA database state is a stored representation of a given EDL specification. Several constraints, that apply to the EARA database state and are related to the ETL specification of the ADISSA transformation, are mentioned at the end of Section 4.2. The difficulties of modeling the ADISSA transformation (because of these constraints), and the proposed enhancements to the existing ETL (based on the EARA database definition) will be discussed in Chapter 5.

Chapter 5

Proposed Extension/Enhancement to Metaview

In this chapter various enhancements to the existing ETL are proposed. In Section 5.1, the syntax and semantics of ETL are briefly described. In Section 5.2, various difficulties of modeling the ADISSA methodology in ETL are discussed. In Section 5.3, the definition and operations of a *transformation aggregate* are proposed. In Section 5.4, we redefine the semantics of the built-in MAP_OF function of ETL. In Section 5.5, other improvements to the existing ETL are proposed. The final section summarizes the chapter.

5.1 ETL

ETL (Environment Transformation Language) is a language which describes the transformation process between specifications at the environment level in Metaview. An extended BNF description of the first version of ETL can be found in [BST91].

Besides the *fundamental data types* (integer, boolean and string), ETL supports the following data type:

1. *Base type* – A variable of base type may be of entity type, relationship type or aggregate type which is specified in the EDL specification in the source or target environments. Taking the EDL specification of DFD environment as an example. The declaration

   process: p;

   defines a variable which has type process. The scope of a base type variable is limited to the current rule unless the variable is declared globally. The purpose of a base type variable is to bind a variable to an instance in the EARA database state of the same type. This enables us to access/create instances in the EARA database state.

2. *Transformation attribute type* – A transformation attribute is an attribute which is of the fundamental data type that is added to the base type. The result is as if we have added a new attribute to the original EDL specification. This enables us to add more programming features to the EDL specification without destroying the integrity of or over complicating the environment definition. For example, control data flow (CDF)

is a special kind of data flow but it does not exist in the source DFD environment until we perform the ADISSA transformation. Therefore we can declare a transformation attribute to the entity type *dataFlow* to indicate whether the data flow is CDF:

TRANS ATTR BOOLEAN OF dataFlow: cdf ← FALSE;

The above statement initializes the cdf attribute of every instance of dataFlow to FALSE indicating initially no data flow is a control data flow.

3. *SET data type* – Since EARA model is based on the notion of sets and functions, set structure is supported in ETL. However in the first version of ETL, SET is homogeneous in nature, that is, no SET can contain objects of different types.

4. *Array type* – Arrays are supported in ETL. Unlike many programming language, it is not necessary to declare the size of an array in the ETL specification because ETL is a specification language.

ETL uses standard control constructs. IF-THEN and FOR loops constructs are supported. These constructs enable an analyst/designer to recognize certain conditions previous to the execution of actions. There is a built-in function for the mapping of elements from the source environment to the target environment (i.e., the MAP function). The MAP_OF function is used to retrieve an image object from an existing mapping. Oracle intervention is also supported in ETL so that an analyst/designer can participate in the transformation process.

Appendix A provides an extended BNF description of the basic ETL, plus the proposed extensions which are discussed in the remainder of this chapter.

5.2 Difficulties of Modeling ADISSA Transformations in ETL

The ADISSA transformation converts a data flow diagram into a series of finite state machines. During the ADISSA transformation, the following intermediate structures are created:

1. ADISSA-transactions

2. Interpretations

3. FSM-transactions

Identification of ADISSA-transactions is the first step of the ADISSA transformation. Next, each ADISSA-transaction is modified to incorporate the notion of interpretation and converted into an FSM-transaction. Finally, each FSM-transaction is mapped to a finite state machine. Each of these intermediate structures need to be represented using ETL and used in the ETL specification of the ADISSA transformation.

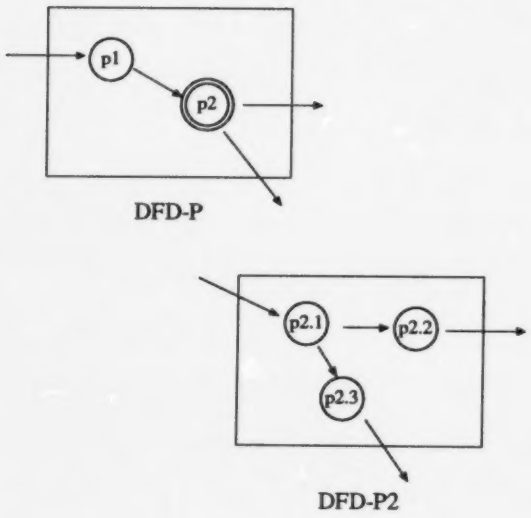


Figure 5.1: Example of process explosion

5.2.1 Keeping the Source Specification Intact

Since the ADISSA transformation modifies the source specification during the transformation process, e.g., rule I5 (Section 2.2.3.1) add a new CDF instance and a new relationship instance into the specification database. It is desirable to keep the source specification intact. We assume the ADISSA transformation process works on a copy of the source specification, leaving the source specification unchanged.

5.2.2 Identification of ADISSA-transactions

As mentioned in Section 2.2.2, identification of ADISSA-transactions involves partitioning the set of all *elementary* processes in the data flow diagram. It is accomplished by defining an equivalence relation on the set of all elementary processes such that if p_1 and p_2 are elementary processes, then $p_1 \equiv p_2$ if and only if there is a data flow connecting p_1 and p_2. It is difficult to determine whether two elementary processes are connected by a data flow in the EDL specification. The reason is if two elementary processes belong to different aggregates, no relationship can be defined between them (Section 4.2). For example, consider Figure 5.1 in which the process p_2 can be exploded into a more detail data flow diagram DFD-P2. In the EDL specification of Figure 5.1, processes p_1 and $p_{2.1}$ are not connected by any data flow even though they are conceptually. In the EDL specification, DFD-P and DFD-P2 are stored as separate aggregates in the EARA database. Process p_2 is an entity of type *processDecomposition* (Section 4.3.1) and is connected to p_1 via the relationship *sends* and is the only entity that is connected to p_1. The situation may get complicated if there is another entity of type *processDecomposition*, say p_3, which can be exploded into DFD-P2 as well (Figure 5.2). In this case there are two distinct copies of $p_{2.1}$ connected to p_1 conceptually. The *aggregate identification function*, *ai*, allows an aggregate, which is composed of many entities and relationships, to be treated as a single identifiable entity within another aggregate. It is not necessary for the latter entity and aggregate to be

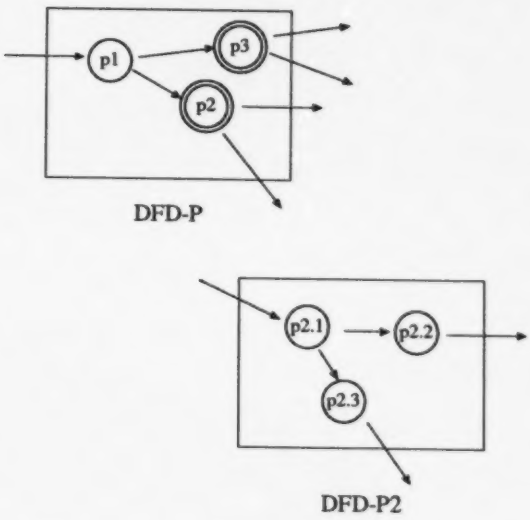


Figure 5.2: Processes p_2 and p_3 explode into the same DFD-P2

unique. Process p_2 and p_3 in Figure 5.2 are such examples.

Even though the DFD environment is formalized in the EARA model, in practice the DFD environment itself is most often used informally. For example, correspondence is often not clearly established between the data flows going in/coming out of the parent process and the data flows going in/coming out of the child DFD. Designers usually establish correspondence based on the naming of the data flows. Consider Figure 5.3 which is copied from Figure 2.2. The data flow *b* is going from the data store D_1 into f_1 in DFD-F. When process f_1 is expanded into DFD-F.1, there must be a data flow coming from a data store, across the boundary and into the diagram DFD-F.1. In both DFDs, the correspondence is done by using the same name *b*. When the two data flows are named differently, designers will usually assign special names to the data flows so that correspondence can be established manually. For example, the data flow *b* in DFD-F could be named *library data* and the data flow *b* in DFD-F.1 could be named *library info*. Designers can still establish the correspondence in this case. When DFD environment is formalized in the EARA model, such a correspondence cannot be established automatically and designer input is needed. For example correspondence between the data flows in DFD-P and DFD-P2 of Figure 5.2 cannot be established automatically.

For the purpose of formalizing the ADISSA transformation, we assume we begin with a flat data flow diagram because conceptually every hierarchical DFD can be decomposed into a flat DFD. For example, a flat DFD for Figure 5.2 might be like Figure 5.4. Based on the EDL specification of data flow diagram environment (Section 4.3.1), the flat data flow diagram will be represented by the the following EDL specification:

AGGREGATE_TYPE data_flow_aggregate
  COMPONENTS (tree, process, dataFlow, terminator, dataStore,
    sends, access, connective, associated_tree,
    dataElement, dataElementsInFlow, dataStored)
  ATTRIBUTES (description: text);

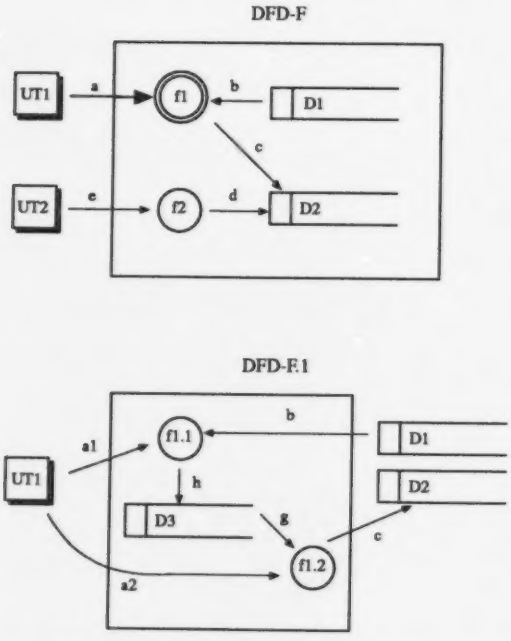


Figure 5.3: Another example of process explosion

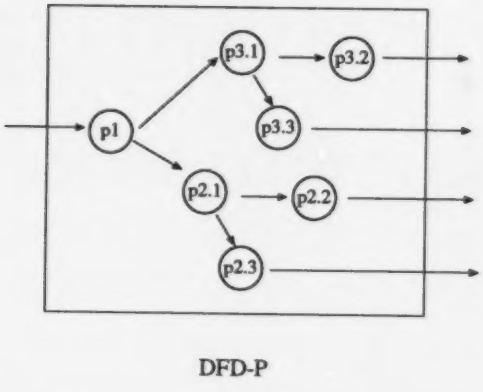


Figure 5.4: After replacing p_2 and p_3

5.2.3 Working with Multiple Environments

In the description of the EARA model, there does not exist a set theoretical definition of environment. When dealing with multiple environments, we run into a problem. The problem is whether an entity type name can be used in different environments and has different attributes in each of the different environments. For example, some authors like to use the term *function* instead of *process* in data flow diagram environment. Babin *et al.* use the term *function* instead of *process* in the original paper of the ADISSA transformation [BLS91]. The notion of *function* is also part of the HOS environment [HZ76]. In the EDL specification of HOS environment, *function* is used as an entity type name [McA88]. If there were an ETL specification which transforms a DFD environment specification into an HOS environment specification and the type name *function* is used in both EDL specifications, it would not be clear which specification is referred to if we declare a base type variable of the type *function*.

In this thesis, we will not attempt to give a definition of environment. We will assume the sets AgT, ET, RT are global to all the environments, that is, if an entity type name or a relationship type name or an aggregate type name is used in one environment, the same type name will not be used in another environment.

5.2.4 Decomposing the ADISSA Transformation into Several Phases

Since the ADISSA transformation is a set of rather complex transformation rules, a natural approach is to decompose the ADISSA transformation into several phases, e.g. the first phase is to identify and construct the ADISSA-transaction, the second step is to construct the interpretations and so on, and each phase is represented as an ETL specification. One problem of this approach is information between different ETL specifications cannot be exchanged. For example, the values of the transformation attributes from a previous phase cannot be passed to the next phase. We need a history of the ETL specifications to be available during the execution of an ETL specification.

The approach we adopt in representing the ADISSA transformation is to represent all the transformation rules in one ETL specification. But it should be pointed that the specification is represented by several rules. If in the future, there is a mechanism of exchanging information between ETL specifications, it may be a good design practice to split the ADISSA transformation into several phases to reduce the size and complexity of an ETL specification.

5.2.5 Multiple Occurrences of Terminators

Another problem is handling the duplication of terminators and data flows. It happens when we apply the rules M1 (Section 2.2.3.2) to construct an FSM-transaction. The result of the rule M1 is that a terminator (or data flow) can have several occurrences with the same name as the original. The EARA model does not allow multiple occurrences of entity or relationship instances. An entity instance is uniquely identified by its name and the name of the aggregate of which it is a component. We cannot have two distinct entity instances, which have the same name, to be the components of an aggregate instance.

By carefully reviewing the rules of M1 and M4, we can discover that if a terminator has n outgoing CDFs and m incoming CDFs, after applying the rules M1 and M4, it will have

- $n + 1$ distinct occurrences if $m \neq 0$. One occurrence has no outgoing CDFs (i.e., it is in the ending set.), and each of the other occurrences (it may not exist if $n = 0$) has exactly one outgoing CDF.
- n distinct occurrences if $m = 0$. Every occurrence has exactly one outgoing CDF.

Therefore we can simplify the rules M1 and M4 by applying the above conclusion to the ETL specification of the ADISSA transformation.

5.2.6 Capturing Intermediate Structures

Each of the ADISSA-transactions, interpretations and FSM-transactions can be considered a separate data flow diagram. In EARA terms, each of the intermediate structures is a set containing different types of entity and relationship instances. There are two ways to proceed in representing these structures using the first version of ETL:

1. Make use of SET structures.
2. Save each intermediate structure as an aggregate.

The first method is not desirable because it creates complicated data structures and makes the ETL specification hard to understand. SET data types in the first version of ETL allows only entity/relationship instances of the same type to be grouped. Therefore, in order to represent what entity instances are components of an intermediate structure, the following data structures are needed: SET OF *terminator*, SET OF *dataFlow*, SET OF *dataStore* and SET OF *process*. This is not enough to represent an intermediate structure because we need to know what relationships hold among the entities. The EARA model does not treat a relationship to be simply a cartesian product of sets of entity instances. Instead, a *relationship instance* is created and is associated with a set of finite lists of entity or aggregate instances (See the *rpi* function in Section 4.2). Therefore we also need SET OF *sends*, SET OF *changes*, SET OF *reads*, and so on, to capture the relationship instances.

In the second approach, each intermediate structure can be saved in the specification so that information can be retrieved later. When an intermediate structure is saved as an aggregate, it means a new aggregate instance is created in the EARA database state. If an entity or a relationship instance in the "flat" DFD specification is a part of the intermediate structure, a new entity or relationship instance is created with the same name that must belong to a new aggregate instance. We cannot create another aggregate instance and let an existing entity or relationship instance be a component of the new aggregate instance because every entity or relationship instance is the component of one and only one aggregate instance. Therefore, new entity or relationship instance must be created. In order to establish the correspondence between the new and the original instances, both instances have the same name. It does not violate any constraint because an entity instance is uniquely identified by its name and the aggregate instance of which it is a component.

In terms of the number of operations or steps to create new entity and relationship instances, this method substantially increases the work of the designer. Moreover, the bigger the number of operations or steps is, the easier a designer will make mistakes in writing ETL code. Since ETL is a specification language, the ETL specification should not be so complicated that the abstraction of the transformations is lost or hard to comprehend.

To solve the problem we introduce the notion of a *transformation aggregate* which behaves like an aggregate and a set at the same time. It adds more descriptive power to the current existing ETL. The definition and operations of transformation aggregate are discussed in the Section 5.3.

5.2.7 MAP Function

In the first version of ETL, the built-in function MAP creates a correspondence between an entity in the source environment and an entity in the target environment. When an entity can have more than one image in the target environment, the syntax of the MAP becomes unclear. For example, in the first version of ETL, we map an entity, say t, to another entity, say r, via the following ETL code:

(1) MAP t TO r;

We can also map t to more than one entity via the following ETL code:

(2) MAP t TO r, s;

Statement (2) works only if we know in advance t is mapped to r and s. Suppose during the execution of the transformation, we have established the mapping from t and r via statement (1) and later we find that s is also an image of t. If we write

MAP t TO s;

it is not clear whether we want to erase the previous image of t and remap t to s, or we want to add r to the existing image of t just like statement (2). The same situation may happen to the ADISSA transformation. A terminator or a data store can have more than one image in the target environment because it can belong to several ADISSA-transactions. We may not know in advance how many images it can have.

To solve the problem, we redefine the semantics of MAP_OF function in Section 5.4 and eliminate the use of MAP function.

5.3 Definition and Operations of Transformation Aggregate

This section is a continuation of Section 5.2.6. The intermediate structures (e.g. ADISSA-transactions, interpretations of an ADISSA-transactions, and FSM-transactions) are needed to be represented by some data structures, other than SET data types or new aggregate instances, so that the ADISSA transformation can proceed. As mentioned at the end of Section 5.2.6, using new aggregate instance to represent the intermediate structures substantially increases the work of the designer. A natural approach will be to allow a SET data type to be heterogeneous so that any entity or relationship can be in the SET data structure.

Consider Figure 5.5 (copied from Figure 2.6) which is an example of an ADISSA-transaction. Based on the EDL specification of DFD (Section 4.3.1), Figure 5.5 can be described as follows:

1. one entity instance of type *terminator*, i.e., UT3,
2. one entity instance of type *process*, i.e., $f1$,

Figure 5.5: An example of ADISSA-transaction

3. one entity instance of type *dataStore*, i.e., D1,
4. two entity instances of type *dataFlow*, i.e., a and g,
5. one relationship instance of type *sends*, and the associated finite list of participants is (f1,a, UT3),
6. one relationship instance of type *reads*, and the associated finite list of participants is (f1,g, D1).

If a heterogeneous SET data structure is used to represent Figure 5.5, all the above instances (i.e., five entity instances and two relationship instances) have to be in the data structure. Some observations can be made if such a heterogeneous SET data structure is available:

1. not every arbitrary heterogeneous SET data type is useful to a designer. A designer wants to know not only what entity instances are in the data structure, but also what relationships hold among them.
2. if an entity is a participant of one of the two relationship instances, this entity instance is also a member of the data structure.

Based on the above observations, we introduce the notion of a *transformation aggregate*.

5.3.1 Definition and Properties of Transformation Aggregate

We begin by reviewing the *component mapping function*:

$$cm : (EI \bigcup RI) \rightarrow AgI$$

where EI is a finite set of entity instances, RI is a finite set of relationship instances, and AgI is a finite set of aggregate instances. Given $x \in AgI$, mathematically the set $cm^{-1}(x)$ is the set of all components of the aggregate instance x. A *transformation aggregate of an aggregate* x is a subset $S \subseteq cm^{-1}(x)$ which satisfies the following constraint:

(V1) For each $r \in S \bigcap RI$ and any entity instance e such that e appears in any list $l \in rpi(\{r\} \times RN)$, we must have $e \in S$ where RN is a finite set of role names and rpi is the *relationship participant instance function*

$$rpi : (RI \times RN) \rightarrow \text{powerlist}_1(EI \bigcup AgI)$$

(V1) states that if a relationship instance r is an element of S, its entity instances participants will also be in S.

The constraint (V1) is important. Given a collection of entity instances, a designer may also be interested what relationships hold among the entity instances. As mentioned in Section 5.2.6, the EARA model does not treat a relationship among the entity instances

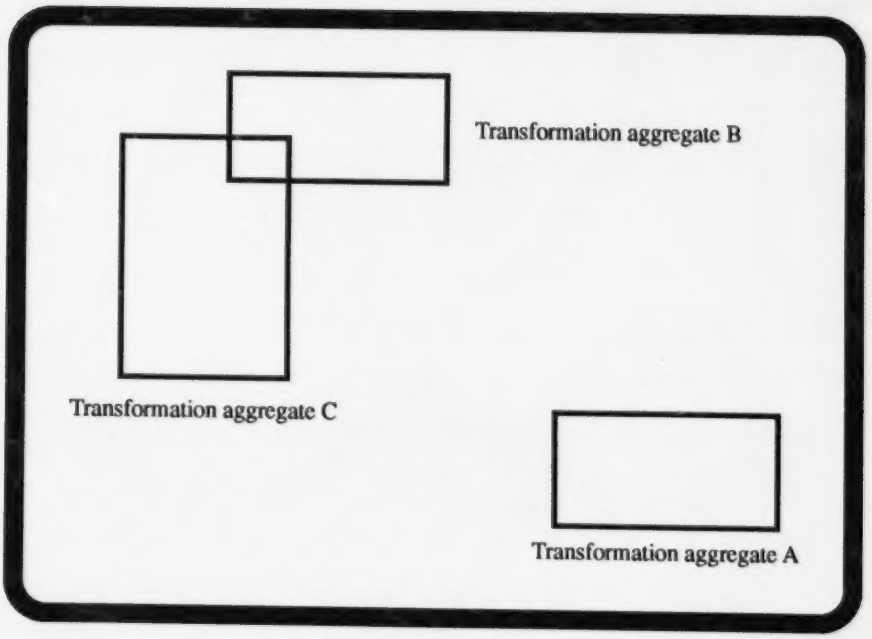


Figure 5.6: Conceptual view of transformation aggregates

simply as a cartesian product of sets of entity instances, instead the relationship is represented by a *relationship instance*, which is associated with a set of finite lists of entity or aggregate instances. (V1) is an abstract way of stating that not only a relationship instance is considered, but also the entity's participants are considered.

A transformation aggregate is comparable to the notion of view in relational database. A transformation aggregate enables the identification of certain subset of the components of an *existing* aggregate without the creation of a new aggregate instance. If we think of an aggregate instance x as the boundary of the rounded-corner rectangle as shown in Figure 5.6 and its set of components $cm^{-1}(x)$ as the interior of this rectangle, a transformation aggregate is a region, which satisfies the constraint (V1). Transformation aggregates have the property that two transformation aggregates can have a non-empty intersection, while this is not not true for two aggregates, i.e., an entity instance can belong to several transformation aggregates but it can belong to only one aggregate instance. For example, the transformation aggregates B and C in Figure 5.6 have a non-empty intersection. If aggregate instances were used, no two aggregates could be allowed to intersect.

Transformation aggregates provide a solution to the problems mentioned in Section 5.2.6 because

1. each of the intermediate structures (ADISSA-transactions, interpretations and FSM-transactions) contains entity/relationship instances of different types,
2. the intermediate structures contain common components, and
3. no new aggregate instances are needed to represent the intermediate structures.

Hence, the intermediate structures are adequately and sufficiently represented by transformation aggregates.

Transformation aggregates materialize during the transformation processes. Section 7.2 addresses the issue whether transformation aggregates should be saved after the transformation.

5.3.2 Manipulation of Transformation Aggregates

To make use of the special nature of transformation aggregates, we extend the existing ETL to allow the manipulation of transformation aggregates with ease and convenience.

1. **Declaration:** Any transformation aggregate must be constructed from an aggregate type much like the way we declare an aggregate variable. Given an aggregate type *AggType*, we declare a transformation aggregate variable *vg* as follows:

   TRANS AGGREGATE OF AggType: vg;

2. **Creation:** If *vg* is declared as above and x is an aggregate of type *AggType*, we use the following construct to bind *vg* to a transformation aggregate of x. (We also say 'bind *vg* to x' and it is understood that we are binding *vg* to a transformation aggregate of x as long as *vg* is declared as a transformation aggregate variable.)

   CREATE TRANS AGGREGATE AggType (vg) WITHIN x;

   The WITHIN clause is necessary because a transformation aggregate can only be constructed from an *existing* aggregate.

3. **Destruction:** If *vg* is bound to a transformation aggregate of an aggregate x, we can use the following statement to unbind *vg*:

   DESTROY TRANS AGGREGATE AggType (vg) WITHIN x;

4. **Scope of a transformation aggregate:** Unless a transformation aggregate is declared as a global variable, its scope is limited to the current rule.

5. **Inserting existing elements into a transformation aggregate:** Since transformation aggregates must satisfy the constraint (V1), care must be taken when adding existing elements into a transformation aggregate. **Inserting a relationship instance into a transformation aggregate will automatically insert all the entity instances participants of that relationship instance into the transformation aggregate.** This means that (V1) is satisfied automatically. If *vg* is a transformation aggregate variable that is bound to an aggregate x and if o is a component of x, we add o to *vg* using the following construct:

   INSERT(o,vg);

   We also allow the addition of a relationship instance by using its type name plus its participants as in the following example:

INSERT(sends(p,df,q), vg);

where *sends* is a relationship type. The resulting effect is the same as adding a relationship instance of type *sends* with participants p, df, q.

6. **Deleting existing elements from a transformation aggregate:** Existing elements can be removed from a transformation aggregate. It should be pointed out the elements still exist in the specification database. Since transformation aggregate must satisfy the constraint (V1), care must be taken when removing an existing element from a transformation aggregate. **Deleting an entity instance from a transformation aggregate will automatically delete all the relationship instances that contain that entity instance as participant.** This means that (V1) is satisfied automatically. We use DELETE construct to delete an element from a transformation aggregate. Its usage has the opposite effect of the INSERT construct.

7. **Adding an element into a transformation aggregate more than once has no effect:** We are not treating a transformation aggregate as a multiset. Therefore adding an element into a transformation aggregate again will not create multiple occurrences of that element.

8. **Finding the aggregate:** Any transformation aggregate must be constructed from an existing aggregate. It is a well-defined mapping to map a transformation aggregate to the aggregate from which the transformation aggregate is constructed. Given a transformation aggregate variable *vg* which is bound and an aggregate variable *agg* which is unbound, we can bind *agg* to the aggregate to which *vg* is bound as follows:

   agg ← LOCATE_AGGREGATE(vg);

   We need this function because

   - we may need to search for some entity or relationship instances that may not be members of a transformation aggregate. In this case we need to search for the instances in the original aggregate. If no such function is available, the name of the aggregate must always be known to the current rule (e.g., by passing the aggregate as a parameter to the current rule). If this situation arises too often, it is not convenient to the implementor who develops the ETL specification.

9. **Creating/destroying database objects via a transformation aggregate is not allowed:** We do not allow users to create/destroy database objects via a transformation aggregate. Users should use `LOCATE_AGGREGATE` to find out the original aggregate and create/delete objects from that aggregate. If the users would like to insert the new object into the transformation aggregate, they should use the `INSERT` statement to do it.

10. **Union and Intersection of transformation aggregates:** It is interesting to note that union of two transformation aggregates of the same aggregate is also a transformation aggregate. The same applies to intersection. Therefore if vg_1, vg_2, vg_3 and vg_4 are transformation aggregates variables bound to the same aggregate, we can perform the following operations:

$vg_3 \leftarrow vg_1$ UNION vg_2;
$vg_4 \leftarrow vg_1$ INTERSECT vg_2;

11. **Accessing elements of a transformation aggregate:** We access elements of a transformation aggregate in exactly the same way we access an aggregate.

5.4 Semantics of MAP_OF Function

There is no relationship that can be defined such that entity participants can come from different aggregates [McA88]. Any relationship and its entity participants must belong to the same aggregate. In the first version of ETL, correspondence between entities from different aggregates was established via the MAP statement. To a certain extent this establishes a relationship with entity participants from different aggregates. The limit of the MAP statement is we can only establish a correspondence between entities from *two* different aggregates.

For the purpose of this thesis, we assume the source specification consists of only one aggregate and the target specification can consist of more than one aggregates. Let x be an aggregate of the source environment and let $y_1, y_2, \ldots, y_n$ be aggregates of the target environment. Let

$$\begin{aligned} E_s &= EI \bigcap cm^{-1}(x) \\ E_{ti} &= EI \bigcap cm^{-1}(y_i) \quad \text{for } 1 \leq i \leq n \end{aligned}$$

where cm is the component mapping function, EI is the set of entity instances. Transformation from x to $y_1, y_2, \ldots, y_n$ is a mapping

$$MAP\text{: } E_s \rightarrow \text{powerset}(E_{t1}) \times \text{powerset}(E_{t2}) \times \cdots \times \text{powerset}(E_{tn})$$

Given an entity e in the source aggregate x, the image of e under the transformation MAP is a collection of subsets of entities where each subset comes from each target aggregate. The powerset in the definition of MAP is necessary.

If $e \in E_s$ and has no corresponding image in the target aggregate y_i, then the i-th place of $MAP(e)$ will be $\emptyset$. In particular if $MAP(e)$ has no image in any target aggregate,

$$MAP(e) = (\emptyset, \emptyset, \ldots, \emptyset).$$

To model the mapping MAP in ETL, we eliminate the MAP statement and keep the MAP_OF statement. The MAP_OF statement must be used together with the WITHIN statement. Given $e \in E_s$, the ETL statement

MAP_OF(e) WITHIN y_i

retrieve the image set of e in E_{ti}. *Unlike the first version of ETL, MAP_OF statement returns a set rather than a single element.*

- Initially MAP_OF returns the *empty set* because correspondence between elements are not established yet.

To test whether a set or a transformation aggregate is empty or not, we add a new built-in function **IS_EMPTY**. It returns **TRUE** if the given set or transformation aggregate is empty; otherwise it returns **FALSE**.

If $o \in E_{ti}$ is an image of $e \in E_s$, we establish the correspondence using the following ETL codes:

INSERT(o, MAP_OF(e) WITHIN y_i);

because MAP_OF returns a set data type.

5.5 Other Enhancements to ETL

The following enhancements are added to the first version of ETL. The enhancements can be classified into two areas:

- to clarify the syntax and semantics of ETL.
- to increase the nonprocedural aspect of ETL, and

Enhancements to clarify the syntax and semantics of ETL:

1. **WITHIN clause:** It is added whenever it is not clear which aggregate applies when referencing an object. An entity or relationship instance cannot exist on its own. According to the constraints on EARA database state, we must specify the name of the aggregate in which the instance is a component. The following ETL statement is allowed in the first version of ETL:

   CREATE ENTITY process (p);

   When working with multiple aggregates, it may not be clear in which aggregate we want to create an instance of type *process*. To avoid ambiguity, we add a **WITHIN** clause to the above statement:

   CREATE ENTITY process (p) WITHIN agg;

   where *agg* is an aggregate variable.

2. **ELSE clause:** The first version of ETL contains only the IF-THEN construct, ELSE clause is added to make the construct more expressive.

3. **TYPE_OF clause:** Given an entity instance, return its type. In the EDL specification of DFD environment, the participants of a *send* relationship instance for the roles *source* and *destination* can be either a *process* instance or a *terminator* instance. When we search the database by locating all the *sends* relationship instances, we may want to know the types of the participants.

4. **Use of role names:** The use of role names in a relationship is allowed to perform query or access the participants.

5. **DESTROY clause:** This is the opposite of the **CREATE** clause. In the first version of ETL, we can create aggregate/entity/relationship instance in the target environment. Sometimes it is desirable in ETL to delete an database object.

Enhancements to increase the nonprocedural nature of ETL:

1. **FIND clause and NULL value:** When a base variable is declared, its initial value is set to **NULL**. A **FIND** clause is added to find the *first* entity instance or relationship instance that satisfies certain conditions. In the first version of ETL, there is no easy way to find the first entity instance or relationship instance that satisfies certain conditions. In addition there is no way we can know whether the search is successful or not. For example to find the first *dataStore* instance that matches a given name, in the first version of ETL one can use the following ETL codes:

```
found ← FALSE;
FOR EACH ds IN SET OF dataStore WITHIN agg WHERE ds.name = givenname
AND found = FALSE
    *** If there is one, set the boolean variable found to TRUE so that
    *** we can exit the FOR loop earlier.
    found ← TRUE;
ENDFOR;
IF found = FALSE
    *** We cannot find it.
ENDIF;
IF NOT found = FALSE
    *** In the first version of ETL, there is no ELSE construct.
    *** We find it.
ENDIF;
```

The above ETL code works except it is highly procedural. With **NULL** and **FIND**, we can modify the above code to:

```
ds ← NULL;
FIND ds IN SET OF dataStore WITHIN agg WHERE ds.name = givenname;
IF ds = NULL
    *** We cannot find it.
ELSE
    *** We find it.
ENDIF;
```

2. **EXTERNAL clause:** When there are situations that cannot be handled by ETL, we rely on some external routines to handle the situations. In this case, we use an **EXTERNAL** clause to invoke an external routine. Taking the ADISSA transformation as an example, the following situations cannot be handled easily by the ETL:

   (a) Selecting components to construct an interpretation. Since *how* the components are selected is not relevant, we may use an **EXTERNAL** clause to invoke an external routine to select the components in this case. Such an external routine may use a graphical interface to let the user to select the components.

(b) Rewrite the logical expression into some normal form. To apply the rule R2 (Section 2.2.4.2), we must determine the number of groups of ANDs separated by XOR. If the entered logical expression is not in disjunctive normal form, one must reorganize the expression to find out the number of groups of ANDs. This situation is best handled by some external routines.

5.6 Summary

In this chapter we have discussed various difficulties that arise when we represent the ADISSA transformation using ETL. As a result of seeking solutions to these problems, the notion of a transformation aggregate is proposed, and the semantics of the built-in MAP_OF function is redefined. Various other improvements to the existing ETL are also proposed. The ADISSA transformation will be represented using the proposed ETL and presented in Chapter 6.

Chapter 6

Representing the ADISSA Transformation and Software Metrics in ETL

In this chapter, we represent the ADISSA transformation in ETL. Since the complete ETL specification is long, Section 6.1 provides an outline of the ETL specification of the ADISSA transformation. The full ETL specification of the ADISSA transformation is presented in Appendix B. In Section 6.2, we demonstrate how ETL can be used to compute the metrics discussed in Chapter 3. The last section summarizes the chapter.

6.1 Outline of the ETL Specification of ADISSA Transformation

6.1.1 Assumptions and Overview of the ETL Specification

We assume the source specification satisfies the followings:

1. As mentioned in Section 5.2.1, the transformations work on a copy of the source specification, leaving the original specification intact.

2. As mentioned in Section 5.2.2, we begin with a flat data flow diagram aggregate of type *data_flow_aggregate*.

3. No *dataFlow* entity participates in more than one *sends*, *reads* or *changes* relationships. This is to comply with the constraint C5 of an FSM-transaction (Section 2.2.3). The ETL specification would not produce the correct transformation if this assumption does not hold. Such a constraint can be specified in ECL (Environment Constraint Language) in Metaview [McA88].

The ETL specification of the ADISSA transformations and the proposed SEC metric consists of 24 different rules. The corresponding ADISSA transformation rules and comments about SEC metric are shown in Table 6.1. In addition, the *Location* column of Table 6.1 lists the location of the ETL specification in this thesis. In general, each ADISSA transformation rule is associated with an ETL rule, except that some of the ADISSA rules

are merged into one single ETL rule because they can be simplified. For example, as discussed in Section 5.2.5, rules M1 and M4 can be simplified.

Figure 6.1 provides a structural view of the ETL specification of the ADISSA transformation. If an arrow is directed form rule A to rule B, then rule B is invoked during the execution of rule A.

Every ETL specification contains:

1. name of the transformation
2. identification of the environments
3. declared global elements
4. main set of rules (sequence of rules to be applied)
5. Details of each rule:
   (a) name of the rule
   (b) antecedents
   - definition of elements used in the rule
   - recognition of certain conditions previous to the execution of actions
   - oracle intervention (i.e., analyst/designer participation in the transformation process)

   (c) consequences
   - mapping of entities, relationships, aggregates or attributes from the source environment to the target environment
   - actions to create entities, relationships, aggregates or attributes

6.1.2 Construction of ADISSA-transactions

An array of transformation aggregates `adissa_trans` is declared globally to capture each ADISSA-transaction:

```
ARRAY OF TRANS AGGREGATE OF data_flow_aggregate: adissa_trans;
```

In order to identify each ADISSA-transaction, a boolean transformation attribute is attached to each process entity:

```
TRANS ATTR BOOLEAN OF process: interconnect <- FALSE;
```

This attribute is used to partition the set of all elementary processes. Each component is stored in the transformation aggregate array `adissa_trans` via the following two rules:

```
1 RULE identify_adissa_transaction(agg: data_flow_aggregate);
2   *** Partition a data flow diagram.
3   process: p, q;
4   INTEGER: i;
5   i <- 0;
6   FOR EACH p IN SET OF process WITHIN agg WHERE p.interconnect = FALSE
7     i <- i+1;
```

ADISSA Transformation	ETL Rules	Location
See § below	identify_adissa_transaction	Chapter 6
See § below	find_interconnect_process	Chapter 6
See • below	begin_transformation	Appendix B
See ○ below	reset_trans_attributes	Appendix B
B1–B4	create_adissa_transaction	Appendix B
See ¶ below	apply_sec_metric	Appendix B
See ¶ below	sec_metric	Chapter 6
See ¶ below	find_connect_process	Appendix B
Initiate I1–I5 and M1–M4	create_fsm_transaction	Chapter 6
I1	specify_starting_set	Appendix B
I2	specify_ending_set	Appendix B
I3	specify_boolean_expression	Appendix B
I4	specify_cdf_and_df	Appendix B
I5	add_additional_cdf	Appendix B
See * below	set_entry_point	Appendix B
See * below	trace_parent	Appendix B
M2–M3†	merge_interpretations	Appendix B
M2‡	merge_boolean_expressions	Appendix B
M2‡	merge_tree	Appendix B
M1 & M4	merge_terminators	Appendix B
S1–S5, S7–S10	analyze_fsm_transaction	Appendix B
T1–T6, R1	generate_sequential_transaction	Appendix B
T4, T7	trace_path_with_dataflow	Appendix B
T4, S6, R2	trace_path_with_process	Appendix B

§ Partition the set of elementary processes into components (Section 2.2.2).
• Initiate the transformation for each ADISSA-transaction.
○ Reset some of the transformation attributes to its initial value.
¶ Apply the Cyclomatic Complexity to the ADISSA-transactions (Section 3.4).
* Locate the set of CDFs that are involved in AND-relationships.
† Except 4b of M2.
‡ Only 4b of M2.

Table 6.1: Correspondence of rule sets

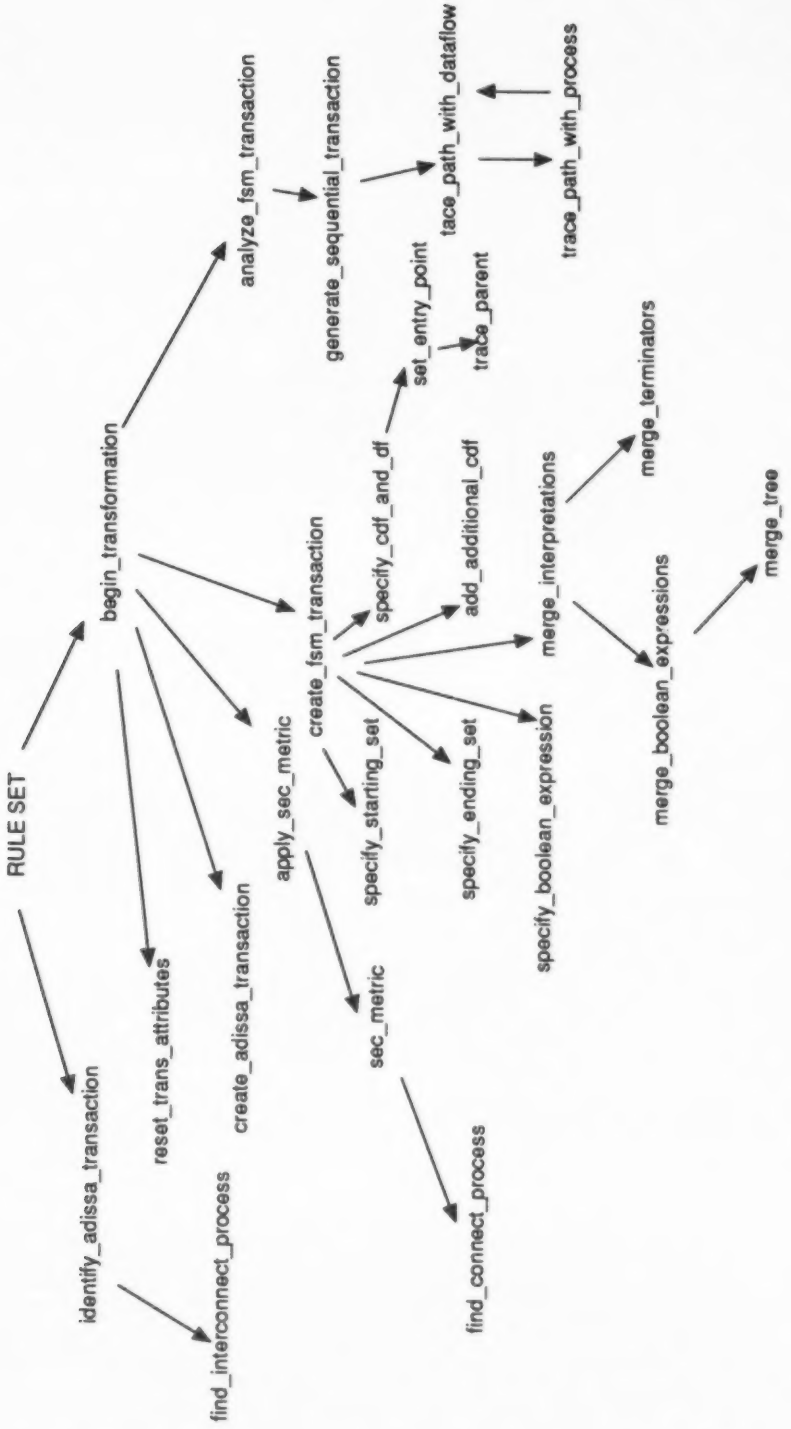


Figure 6.1: Structural view of the ETL specification of the ADISSA transformation

```
 8     CREATE TRANS AGGREGATE (adissa_trans[i]) WITHIN agg;
 9     INSERT(p,adissa_trans[i]);
10     p.interconnect <- TRUE;
11     find_interconnect_process(p,adissa_trans[i]);
12   ENDFOR;
13
14 RULE find_interconnect_process(p: process, adissa: TRANS AGGREGATE OF
15                                                    data_flow_aggregate);
16   *** Find the component that contains p.
17   process: q;
18   data_flow_aggregate: agg;
19
20   agg <- LOCATE_AGGREGATE(adissa);
21   *** Retrieve the original aggregate.
22   FOR EACH q IN SET OF process WITHIN agg WHERE
23   (sends(p,*,q) OR sends(q,*,p)) AND q.interconnect = FALSE
24     q.interconnect <- TRUE;
25     INSERT(q,adissa);
26     find_interconnect_process(q,adissa);
27   ENDFOR;
```

In each iteration (Lines 6–12), one component is found and every process in the component will have the `interconnect` attribute set to `TRUE`. Each transformation aggregate is then extended to include other components by applying the rules B1–B4 (Section 2.2.2), thus creating an ADISSA-transaction.

6.1.3 Applying the SEC Metrics

The SEC metric (Section 3.2) can be applied to the resulting ADISSA-transactions before proceeding with the transformation.

```
 1 RULE sec_metric(df: dataFlow, adissa: TRANS AGGREGATE OF
 2                                       data_flow_aggregate);
 3   process p, q, f;
 4   sends: s;
 5   TRANS AGGREGATE OF data_flow_aggregate: orig;
 6   SET OF process: so, de;
 7   data_flow_aggregate: agg;
 8   dataFlow: dff;
 9   INTEGER: k, m, n;
10
11   agg <- LOCATE_AGGREGATE(adissa);
12   CREATE TRANS AGGREGATE (orig) WITHIN agg;
13   *** Work on the copy ***
14   orig <- adissa;
15   *** Calculate the total number of processes.
16   k <- 0;
17   FOR EACH p IN SET OF process WITHIN orig
```

```
18      k <- k+1;
19    ENDFOR;
20
21    *** Locate the initial and terminal point of the data flow df.
22    FIND s IN SET OF sends WITHIN orig WHERE s.data = df;
23    p <- s.source;
24    q <- s.destination;
25
26    *** remove the instance df and any other relationship instances that
27    *** contain df from the transformation aggregate "orig".
28    DELETE(df,orig);
29    *** reset the transformation attribute "interconnect" for the
30    *** transformation aggregate "orig".
31    FOR EACH dff IN SET OF dataFlow WITHIN orig
32      dff.interconnect <- FALSE;
33    ENDFOR;
34    *** Find the two sets of processes that contain p and q
35    *** respectively.
36    find_connect_process(p,orig,so);
37    find_connect_process(q,orig,de);
38
39    m <- SIZE_OF(so);
40    n <- SIZE_OF(de);
41    ORACLE OUTPUT "Data flow ", df.name;
42    ORACLE "has the following SEC metric value:";
43    ORACLE OUTPUT k,m,n;
```

The above rule is a good example illustrating the advantages of transformation aggregates. We take a copy of the ADISSA-transaction (Line 14) and delete the data flow from the copy (Line 28). This action automatically deletes all the relationship instances containing that data flow as participants. Note that it only removes the instances from the transformation aggregate and does not affect the original database state. The set of all elementary processes are then decomposed into two subsets and the number of elements in each subset is computed. The rule `find_connect_process` (Lines 36–37) will find the set of processes that are connected to the given process. It is similar to the rules `find_interconnect_process` (Line 14 of the rule `identify_adissa_transaction`).

6.1.4 Refinement to FSM-transactions

In the next step we refine each ADISSA-transaction to become an FSM-transaction. The main rule for constructing an FSM-transaction follows.

```
1 RULE create_fsm_transaction(adissa: TRANS AGGREGATE OF
2                                     data_flow_aggregate);
3   INTEGER: num, i;
4   ARRAY OF TRANS AGGREGATE OF data_flow_aggregate: itpt;
5   data_flow_aggregate: dfdagg;
6
```

```
 7   *** Find the original aggregate ***
 8   dfdagg <- LOCATE_AGGREGATE(adissa);
 9   ORACLE OUTPUT "How many interpretations are there for the ADISSA ";
10   ORACLE OUTPUT "transaction?"
11   *** Display the ADISSA-transaction to the user (graphically).
12   EXTERNAL display_trans_aggregate(adissa);
13   ORACLE INPUT num;
14   FOR i <- 1 TO num
15     CREATE TRANS AGGREGATE (itpt[i]) WITHIN dfdagg;
16     ORACLE OUTPUT "Please specify components for the",
17                   i, "interpretation";
18     EXTERNAL prompt_for_interpretation(adissa,itpt[i]);
19     *** After returning, the itpt[i] contains an interpretation
20     *** (subtask). We then apply rules to specify the process
21     *** logic itpt[i].
22     specify_starting_set(itpt[i]);          *** Rule I1 ***
23     specify_ending_set(itpt[i]);            *** Rule I2 ***
24     specify_boolean_expression(itpt[i]);    *** Rule I3 ***
25     specify_cdf_and_df(itpt[i]);            *** Rule I4 ***
26     add_additional_cdf(itpt[i]);            *** Rule I5 ***
27   ENDFOR;
28   merge_interpretations(adissa,itpt);       *** Rule M1-M4 ***
```

In the refinement process, the first step is to obtain all the interpretations of the current ADISSA-transaction from the designer. Another array of transformation aggregates is used to store each interpretation: `itpt` (Line 4). The array `itpt` is declared within the scope of the rule `create_fsm_transaction`. The rules I1–I5 (Section 2.2.3.1) are applied to each interpretation stored in the array `itpt`.

After the rules I1–I5 are executed, the rules M1–M4 are then considered (Section 2.2.3.2). Since all the interpretations are stored as transformation aggregates which intersect with each other, merging the interpretations involves taking the union of all the interpretations with the parent ADISSA-transaction.

6.1.5 Create FSMs

In the last stage, we are ready to create the finite state machines in the target environment. The rules T1–T7, S1–S10 and R1–R2 (Section 2.2.4) are combined and implemented in the following rules (see also Table 6.1):

- `analyze_fsm_transaction`
- `generate_sequential_transaction`
- `trace_path_with_dataflow`
- `trace_path_with_process`

To create the target finite state machines, we traverse the FSM-transaction flows beginning with

DFD Specification Environment	*FSM Specification Environment*
terminator	state, action
dataStore	state, action
process	state, action
dataFlow	entry

Table 6.2: Mapping a DFD to an FSM

1. the elements in the starting set, and
2. each control data flow that is involved in an AND relationship.

In Case 2, the rule R1 is applied to create the *transmission state* in the target finite state machine. A path is traversed until one of the followings happens:

1. we reach another control data flow that is involved in an AND relationship, or
2. we reach any element in the ending set.

In the former case, the *calling state* and the *exit state* are created as described in the rule R2. In the latter case, we finish constructing a finite state machine. Mappings between the source specification and the target specification are also established when we traverse the FSM-transaction. Table 6.2 shows the the correspondence between the entity types in the source and the target specification environments. Note that entities such as processes may have more than one image in the target environment.

6.2 ETL Specification of Metrics Support for ADISSA Methodology

In this section, we present the ETL descriptions of the metrics discussed in Chapter 3.

6.2.1 Metrics in the DFD Environment

The metrics presented in this section may optionally be applied to the source specification before the ADISSA transformation. Based on the values returned by these metrics, the designer can

1. adjust the number of elementary processes in a data flow diagram;
2. locate the process with high incoming and outgoing data activity.

PROC: Number of elementary processes in a DFD

```
RULE num_elementary_process(p_agg: processExplosion);
  INTEGER: procnumb;
  process: p;
  procnumb <- 0;
```

```
  FOR EACH p IN SET OF process WITHIN p_agg
    procnumb <- procnumb+1;
  ENDFOR;
  ORACLE OUTPUT "The number of elementary processes is:";
  ORACLE OUTPUT procnumb;
```

This rule is applied to each data flow diagram. The number of elementary processes is contained in **procnumb**.

DEI: Total number of data elements moving into an elementary process

```
RULE count_input_data_elements(p: process, p_agg: processExplosion);
  INTEGER: inde;
  dataFlow: df;
  dataElementsInFlow: deif;

  inde <- 0;
  FOR EACH df IN SET OF dataFlow WITHIN p_agg WHERE
  sends(*,df,p) OR reads(p,df,*)
    FOR EACH deif IN SET OF dataElementsInFlow WITHIN p_agg WHERE
    deif.flow = df
      inde <- inde + 1;
    ENDFOR;
  ENDFOR;

  ORACLE OUTPUT "Total number of input data elements:", inde;
```

This rule is applied to calculate the total number of incoming data elements of an elementary process. For every incoming data flow of the elementary process, the rule accumulates, in **inde**, the number of *dataElement*.

DEO: Total number of data elements moving out of an elementary process

```
RULE count_output_data_elements(p: process, p_agg: processExplosion);
  INTEGER: outde;
  dataFlow: df;
  dataElementsInFlow: deif;

  outde <- 0;
  FOR EACH df IN SET OF dataFlow WITHIN p_agg WHERE
  sends(p,df,*) OR changes(p,df,*)
    FOR EACH deif IN SET OF dataElementsInFlow WITHIN p_agg
    WHERE deif.flow = df
      outde <- outde + 1;
    ENDFOR;
  ENDFOR;

  ORACLE OUTPUT "Total number of output data elements:", outde;
```

This rule is applied to calculate the total number of outgoing data elements of an elementary process. For every outgoing data flow of the elementary process, the rule accumulates, in **outde**, the number of *dataElement*.

DFI: Total number of data flows incoming into an elementary process

```
RULE count_input_data_flow(p: process, p_agg: processExplosion);
  INTEGER: numindf;
  dataFlow: df;

  numindf <- 0;
  FOR EACH df IN SET OF dataFlow WITHIN p_agg WHERE
  sends(*,df,p) OR reads(p,df,*)
    numindf <- numindf + 1;
  ENDFOR;

  ORACLE OUTPUT "Total number of incoming data flow:", numindf;
```

This rule is applied to calculate the number of incoming data flows of a process.

DFO: Total number of data flows outgoing from an elementary process process

```
RULE count_output_data_flow(p: process, p_agg: processExplosion);
  INTEGER: numoutdf;
  dataFlow: df;

  numoutdf <- 0;
  FOR EACH df IN SET OF dataFlow WITHIN p_agg WHERE
  sends(p,df,*) OR changes(p,df,*)
    numoutdf <- numoutdf + 1;
  ENDFOR;

  ORACLE OUTPUT "Total number of outgoing data flow:", numoutdf;
```

This rule is applied to calculate the number of outgoing data flows of a process.

6.2.2 Metrics in the FSM Environment

The following metrics can be applied after the ADISSA transformation. If an area of high complexity is detected, it should be fedback to the designer so that corrective actions can be applied to the source specification.

ST: The number of states in an FSM

```
RULE count_num_states(fsm: fsm_aggregate);
  state: st;
  INTEGER: count;

  count <- 0;
  FOR EACH st IN SET OF state WITHIN fsm
    count <- count+1;
  ENDFOR;
  ORACLE OUTPUT "The number of states is:", count;
```

This rule is applied to count the number of different states in an FSM.

STI: The number of state transitions into a state

```
RULE count_input_transition(st: state, fsm: fsm_aggregate);
  INTEGER count;
  next_state_function: ns;

  count <- 0;
  FOR EACH ns IN SET OF next_state_function WITHIN fsm
  WHERE ns.output_state = st
    count <- count+1;
  ENDFOR;
  ORACLE OUTPUT "The number of incoming state transitions is:", count;
```

This rule is applied to count the number of incoming state transition of a given state.

STO: The number of state transitions out of a state

```
RULE count_output_transition(st: state, fsm: fsm_aggregate);
  INTEGER count;
  next_state_function: ns;

  count <- 0;
  FOR EACH ns IN SET OF next_state_function WITHIN fsm
  WHERE ns.input_state = st
    count <- count+1;
  ENDFOR;

  ORACLE OUTPUT "The number of outgoing state transitions is:", count
```

This rule is applied to count the number of outgoing state transition of a given state.

McCabe's Cyclomatic Complexity:

```
RULE cyclomatic_complexity_for_fsm(fsm: fsm_aggregate);
  INTEGER: node, edge, num_final, mccabe;
  state: st;
  next_state_function: ns;

  node <- 0;
  num_final <- 0;
  FOR EACH st IN SET OF state WITHIN fsm
    node <- node+1;
    IF st.type_of_state = final
      num_final <- num_final+1;
    ENDIF;
  ENDFOR;

  edge <- 0;
  FOR EACH ns IN SET OF next_state_function WITHIN fsm
```

```
  edge <- edge+1;
ENDFOR;

mccabe <- edge + num_final - node + 1;

ORACLE OUTPUT "Cyclomatic Complexity:", mccabe;
```

This rule is applied to calculate the variation of McCabe's Cyclomatic Complexity as discussed in Section 3.4.

6.3 Summary

In this chapter, we have presented the outline of the ETL specification of the ADiSSA transformation. The ETL specification consists of 24 different rules, and captures all the important aspects of the ADISSA transformation. The ETL descriptions of the metrics discussed in Chapter 3 are also presented in this chapter. These metrics are used before and after the ADISSA transformation.

Chapter 7

Conclusions & Future Work

In this Chapter we provide a summary of the thesis and suggest topics for further research.

7.1 Thesis Summary and Contributions

In this thesis we have investigated the feasibility of software development transformations and the use of software metrics to guide transformations by modeling the ADISSA methodology and its related transformation using the Environment Transformation Language of Metaview.

The need for software transformations and software metrics in software development is identified in Chapter 1. To further examine the support for transformations and the use of metrics in Metaview, we chose the ADISSA methodology and its related transformation as a representative transformation based approach. The ADISSA methodology and its transformation are reviewed in Chapter 2. In the ADISSA transformation, a hierarchical DFD is modified to incorporate the notion of ADISSA-transactions and interpretations. Then FSM-transactions are derived and used in the generation of the final FSMs. Software metrics that can be applied to the ADISSA methodology are discussed in Chapter 3. The metrics we have chosen are concerned with the size and complexity of the products produced in the ADISSA methodology, e.g., counting the number of incoming/outgoing data flows of an elementary process, counting the components size resulting from a single edge cut, and measuring the cyclomatic complexity of a FSM, etc.

The EARA model on which Metaview is based is introduced in Chapter 4. The EARA model for software specification environments supports a metasystem approach to the development of CASE environments. To demonstrate the power of the EARA model, we model the DFD and FSM environments using the environment definition language, EDL. A brief review of the ETL is given in Chapter 5. Various difficulties of modeling the ADISSA transformation using the existing ETL are also discussed. The main difficulty is that ETL lacks the descriptive power to capture some aspects of the ADISSA-transactions, interpretations and FSM-transactions. For example, ETL provides SET structure which is homogeneous and therefore cannot describe each of the above notions in a comprehensive way. Capturing these notions using aggregates is also not convenient. Aggregates are treated as independent objects in the sense that there are no entity or relationship instances that can belong to several aggregates. As a result, new instances of aggregate, entity and relationships must be created to represent the notions. To overcome the problem, we proposed the notion of

transformation aggregate to the existing ETL.

A transformation aggregate is comparable to the notion of view in relational database. Therefore, a transformation aggregate behaves like an aggregate but entity or relationship instances can belong to several transformation aggregates. The relaxation of the encapsulation property of aggregation in a transformation approach is necessary in order to model objects that can partake in transformations between different environments. The dual nature of transformation aggregate allows us to capture and provide transformations for ADISSA-transactions, interpretations and FSM-transactions easily. The definition of and operations on a transformation aggregate are introduced in Chapter 5. Various other enhancements to the existing ETL are discussed in the same chapter.

Finally the ADISSA transformation and the metrics discussed in Chapter 3 are represented using the proposed, extended ETL in Chapter 6.

The main contributions of this thesis are in three areas:

1. The feasibility of supporting transformations using a metasystem definition coupled with the rules expressed in a transformation language (ETL) has been established for a rather complex transformations method that exists in the literature (i.e., the ADISSA methodology). The ADISSA methodology does not assume a metasystem approach to software development. Fundamental to the ADISSA methodology is a set of rather complex transformation rules which are well-defined to provide guidelines of transforming DFD specifications into FSM specifications. There is nothing in the literature which reports on the feasibility of applying a metasystem approach to automated transformations. The general feasibility of this approach, while not proven, appears to be promising, given the variations and relative complexities of the transformations required in the ADISSA approaches. The results are encouraging and support the continuation of the metasystem approach to automated transformations using ETL as a representation vehicle.

2. The use of software metrics to guide transformations was examined. In particular, several metrics were described that could be used in conjunction with the ADISSA transformation methodology. The ADISSA methodology was not developed with software metrics in mind to support software measurement activities. The results of this investigation indicate that the coexistence of software metrics and software transformation in Metaview is feasible, in the sense that transformations between specifications can be assisted by software metrics.

3. A set of extensions to ETL was proposed to support transformational activities; namely, transformation aggregates, new MAP_OF function, **EXTERNAL** statements, and other extension discussed in Section 5.5. The approach adopted in this thesis was to take the first version of ETL to represent the ADISSA transformation method, and to describe some metrics that could coexist with the ADISSA transformation method. As a result of this investigation, an improved version of ETL was proposed. The enhancements to the existing ETL were in four areas:

   (a) The descriptive power of ETL was increased. A transformation aggregate was proposed to allow certain subsets of the components of an aggregate instance to be specified without using SET data type or creating new aggregate instances to represent the subsets that were of interest to the designer.

(b) The nonprocedural nature of ETL was increased. For example, **EXTERNAL** clause, **FIND** statement were added to the existing ETL.

(c) The syntax of ETL was clarified. For example, **WITHIN** clause was added whenever it was not clear which aggregate applied when referencing an object.

(d) The semantics of the mapping function was redefined. We learned that an entity instance in the source specification can have more than one image in the target specification. Moreover, we learned that it is often impossible to know in advance how many images of an entity instance in the source specification can have in the target specification. As a result, the semantics of the mapping function was redefined to handle this observation.

We have also learned that the semantics of transformations are not simple. The ETL specification of the ADISSA transformation required 747 lines of code (excluding comments and blank lines) contained in 24 different rules.

7.2 Future Work

As mentioned in Section 5.3.1, transformation aggregates materialize during the transformation processes. If transformation aggregates are required to be persistent, the EARA database state definition can be extended to incorporate the notion of a transformation aggregate. Whether such an extension is necessary depends on the future investigation of other methodologies, other than the ADISSA methodology. If such an extension is desirable, a transformation aggregate can be a part of the specification database. It also provides a way to save any important information gathered during transformation processes so that a designer can retrieve the information later. Moreover, the ETL specification of the ADISSA transformation can be decomposed into several phases. As mentioned in Section 5.2.4, the ADISSA transformation is represented by one ETL specification because there is no known mechanism of retrieving history of other ETL specifications. If a transformation aggregate is a part of the specification database, it provides a mean of exchanging information between different ETL specifications. There are advantages and disadvantages of saving the transformation aggregates after the transformation process is completed.
Advantages:

1. Transformational informational can be retrieved after the transformations.

2. Incremental transformation may be feasible in the sense that if part of the source specification is modified, only the modified portion is being transformed again.

Disadvantages:

1. Since the source and the target specifications are not *static* (i.e., the designer may modify them after a transformation is completed), any information saved in the previous transformation process may not be valid with respect to a new source or target specifications. This gives rise to the problem of trying to maintain valid transformation aggregates relative to source or target specifications.

A possible extension to the definition of the EARA database state to incorporate the notion of a transformation aggregate.

AgI – a finite set of aggregate instances.
EI – a finite set of entity instances.
RI – a finite set of relationship instances.
TAI – a finite set of transformation aggregate instances.
V – a finite set of values.
$agt : AgI \rightarrow AgT \backslash ag(AgT)$
– the **aggregate type** function where
\ is the set difference operator.
$et : EI \rightarrow ET \backslash eg(ET)$
– the **entity type** function.
$rt : RI \rightarrow RT \backslash rg(RT)$
– the **relationship type** function.
$vt : V \rightarrow VT$
– the **value type** function.
$vm : ((AgI \cup EI \cup RI) \times AN) \rightarrow V$
– the **value mapping** function (partial).
$rpi : (RI \times RN) \rightarrow \text{powerlist}_1(EI \cup AgI)$
– the **relationship participant instance** function.
$cm : (EI \cup RI) \rightarrow AgI$
– the **component mapping** function.
$taa : TAI \rightarrow AgI$
– the **transformation aggregate association** function.
$tam : TAI \rightarrow (EI \cup RI)$
– the **transformation aggregate component** function.

Compared to the original definition of the EARA database state, the new definition contains a set of *transformation aggregate* instances, two new functions, i.e., the *transformation aggregate association* function (*taa*) and the *transformation aggregate component* function (*tam*). Since a transformation aggregate must be constructed from an *existing* aggregate, the function *taa* establish the association. The function *tam* establishes the components of a transformation aggregate. Besides the constraint (V1) (Section 5.3.1), one more constraint is needed. For the purpose of completeness, the two constraints are listed below.

(U1) For each $v \in TAI$, $tam(v) \subseteq cm^{-1}(taa(v))$.

(V1) For each $v \in TAI$ and each $r \in tam(v) \bigcap RI$ and any entity instance e such that e appears in any list $l \in rpi(\{r\} \times RN)$, we must have $e \in tam(v)$ where RN is a finite set of role names and *rpi* is the *relationship participant instance function*

Constraint (U1) states that a transformation aggregate of an aggregate instance x is a subset of the components of x.

Besides extending the EARA database state, alternative ways of saving the transformational information can also be considered. The information can be kept with the source specification, with target specification, or even with the transformation process itself. Based on the research of this thesis, it is not clear which of these alternatives is the best and further examination of this problem is necessary.

The correctness of the proposed ETL specifications has to be established by imposing pre-conditions and post-conditions on the source and target specifications. As mentioned

in Chapter 1, ECL (Environment Constraint Language) can potentially solve this problem, but the feasibility of using ECL to prove correctness is still unknown and requires future work.

The difficulties of implementing the ADISSA methodology, as mentioned in Chapter 5, are not completely resolved, e.g., identification of ADISSA-transactions, definition of environment, and working with multiple environments. They need to be addressed. In order to identify the ADISSA-transactions and perform the remaining parts of the transformation, the ETL specification of the ADISSA transformation assumes we begin with a "flat" DFD aggregate of type *data_flow_aggregate*, which is generated from the original hierarchical DFDs specifications. This may not be a satisfying solution to the problem. It may not be obvious how the flat DFD and the original hierarchical DFDs specifications are *related*. For example, is any change to the flat DFD automatically reflected in the original specifications? Is the flat DFD an independent copy of the original specifications? The following problems may require attention when the ETL specification of the ADISSA transformation is to be executed.

1. There are three places where the flat DFD is modified during the ADISSA transformation:

   (a) specifying additional CDF if the original DF is in the opposite direction of the control flow,

   (b) adding a logical expression to the outgoing data flows of a process, and

   (c) duplicating a terminator.

   If the modifications are to be reflected in the original hierarchical DFDs specifications, it may not be easily done.

2. When the SEC metric detects an area that needs to be revised, this area may not be easily located in the original hierarchical DFDs specifications. The elementary processes in an ADISSA-transaction may come from several different data flow diagrams in the source specifications.

3. The mapping is from the flat DFD specification to the target FSM specifications instead of from the original specifications.

The first problem suggests an additional problem. Suppose the source specifications should remain intact during the transformations, and the transformation rules modify the source specifications. Without making backup copies of the source specifications, how can the source specifications remain intact? How can the designer retrieve any information gathered during the transformation process? This can be a difficult problem.

Since a transformed specification is derived from the source specification via the transformation rules, any metric applied in the transformed specification may be possibly transformed backward to a new metric to apply to the source specification. The advantage of applying this new metric is that the designer can estimate in advance the metric which is supposed to apply in the transformed specification. For example, the metric STI, which computes the number of state transition into a state, can be computed in advance even before the transformation. It is equal to the number of incoming CDFs into the inverse element of the state of the transformation mapping function, if it has one. When an area of

high complexity is detected in the target specification, it may not be desirable to change the original specification and restart the transformations. But if such an area can be predicted prior to the transformation, the designer can take significant corrective actions earlier in the software life cycle. Since the semantics of transformations are not simple, it is not known whether such a reversing of a metric is always possible.

Implementation of ETL is a key issue in the future work. A compiler must be completed for the proposed version of ETL. The enhancements to ETL proposed in this thesis are based on examining the ADISSA methodology. Undoubtedly, newer versions of ETL will appear in the future as a result of investigating further transformational and metric activities.

Another research direction is reverse transformations. ADISSA methodology is a forward transformation methodology from the analysis specifications to the design specifications. If modifications are made in the target FSM specifications, the corresponding changes in the DFD specifications may not be obtained easily by reversing the transformation. In particular, when designer input is needed to perform the forward transformation, (i.e., via ORACLE statements in ETL), such a reverse transformation may not be possible. In general, the problem of reverse transformations falls within the study of reverse engineering. The main vehicle for achieving reverse engineering is the abstraction of code into design concepts [BC86]. Reverse engineering requires that we can work backward to specification and then forward, through redesign, to a better implementation. This can be difficult, however, when clues to the design are scattered in threads throughout the implementation, or when optimization in the code obscures the design [ABFP86]. Previous maintenance can also hide the skeleton of the original design.

The study of automated transformations is a new area. We believe in order to discover more aspects of the transformational activities, the best approach is to implement other formal transformation methodologies using our metasystem approach. The first version of ETL was developed to support the transformations from DFD specifications to SC (Structured Chart) specification [BST91]. As a result of examining the ADISSA methodology in this thesis, the first version of ETL is improved by introducing the notion of a transformation aggregate, plus some other enhancements proposed in Chapter 5. The next logical step is to continue implementing the other formal transformation methodologies, e.g. [Sac92], to discover additional aspects of the transformational activities to improve the ETL.

The metrics discussed in this thesis are related to products. These products, e.g., specifications, and source code, are usually stored in the CASE repository and therefore data can be collected without great difficulty. But some other on-workbench and off-workbench data are difficult to collect, e.g., session length (how much time is spent on writing source code each day, etc), discussion, and thinking. In order to collect these data for process metrics, process models must be integrated into Metaview. An investigation of integration of process models into the Metaview environment is currently under research [BST92a]. Tate *et al.* [TVJ92] have proposed a model which integrates CASE, metrics, and process model enaction.

Another possible future research area is the investigation of integration of case-based reasoning [RS89] and software metrics into Metaview. It has been stressed in Chapter 1 and Chapter 3 that software metrics can aid the designers in decision making and in reminding the designers to take corrective actions earlier in the software life cycle. But the designers may not benefit from software metrics if they want to know *how* to solve/correct the problem. Case-based reasoning is an analogical reasoning method. It means reasoning from old cases or experiences in an effort to solve problems. It has been reported that case-based

reasoning can aid designers in decision making and solving problems [Kol91]. There are two styles of case-based reasoning: problem solving and interpretive. In the *problem solving style* of case-based reasoning, solutions to new problems are derived using old solutions as a guide. Old solutions can provide "almost-right" solutions to new problems, or can assist in providing warnings of potential mistakes or failures. In the *interpretive style* new situations are evaluated in the context of old situations. Interpretive case-based reasoning is often used to evaluate the pros and cons of a problem solution.

The first step of integrating case-based reasoning and software metrics into Metaview is to address the following two problems:

1. Cases representation – How cases are to be represented?

2. Cases indexing – How can we find the right cases?

We believe both problems are not easy and provide important future research directions in software metrics in Metaview.

Bibliography

[ABFP86] G. Arango, I. Baxter, P. Freeman, and C. Pidgeon. TMM: Software maintenance by transformation. *IEEE Software*, 3(3):27–39, 1986.

[Bab89] G. Babin. Implantation exacte de transactions à l'aide de machines séquentielles. Master's thesis, Université de Montréal, April 1989.

[BC86] R. N. Britcher and J. J. Craig. Using modern design practices to update aging software system. *IEEE Software*, 3(3):16–24, 1986.

[Ber73] C. Berge. *Graphs and Hypergraphs*. North-Holland, 1973.

[BF90] Martin E. Bush and Norman E. Fenton. Software measurement: A conceptual framework. *Journal of Systems and Software*, 12:223–231, 1990.

[BLS91] Gilbert Babin, François Lustman, and Peretz Shoval. Specification and design of transactions in information systems: A formal approach. *IEEE Transactions on Software Engineering*, 17(8):814–829, August 1991.

[Boe76] B. W. Boehm. Software engineering. *IEEE Transactions on Computers*, 25(12):1226–1241, December 1976.

[Boe88] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 31(5):61–72, 1988.

[BST91] Germinal Boloix, Paul G. Sorenson, and J. Paul Tremblay. On transformations using a metasystem approach to software development. Technical Report TR 91-19, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, November 1991.

[BST92a] Germinal Boloix, Paul G. Sorenson, and J. Paul Tremblay. Process modeling using a metasystem approach to software specification. Technical Report TR 92-11, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, September 1992.

[BST92b] Germinal Boloix, Paul G. Sorenson, and J. Paul Tremblay. Software metrics using a metasystem approach to software specification. Technical Report TR 92-04, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, April 1992.

[CDS86] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. Benjamin/Cummings, 1986.

[Che76] P. Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[DeM78] T. DeMarco. *Structured Analysis and System Specification.* Yourdon Inc., 1978.

[DeM82] T. DeMarco. *Controlling Software Projects.* Yourdon Press, New York, 1982.

[GS79] C. Gane and T. Sarson. *Structured System Analysis: Tools and Techniques.* Prentice-Hall, 1979.

[Hal77] M. H. Halstead. *Elements of Software Science* North-Holland, 1977.

[HK81] S. M. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, pages 510–518, September 1981.

[HS90] Sallie Henry and Calvin Selig. Predicting source-code complexity at the design state. *IEEE Software*, pages 36–44, March 1990.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[HZ76] M. Hamilton and S. Zeldin. Higher order software - a methodology for defining software. *IEEE Transactions on Software Engineering*, 2(1):9–32, March 1976.

[Kol91] Janet L. Kolodner. Improving human decision making through case-based decision aiding. *AI Magazine*, 12(2):52–68, 1991.

[McA88] Andrew J. McAllister. *Modeling Concepts for Specification Environments.* PhD thesis, Department of Computational Science, University of Saskatchewan, Saskatoon, Saskatchewan, Canada, March 1988.

[McC76] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, pages 308–320, December 1976.

[MDC86] H. Mills, M. Dyer, and P. A. Currit. Certifying the reliability of software. *IEEE Transactions on Software Engineering*, 12(1), 1986.

[MDL87] H. Mills, M. Dyer, and R. Linger. Cleanroom software engineering. *IEEE Software*, 4(5):19–25, September 1987.

[MLH87] H. Mills, R. C. Linger, and A. R. Hevner. Box structured information systems. *IBM Systems Journal*, 26(4), 1987.

[MM85] James Martin and Carma McClure. *Diagramming Techniques for Analysts and Programmers.* Prentice-Hall, 1985.

[Odo90] J. E. Odorn. Using box structures for definition of requirements specifications. *IBM Systems Journal*, 29(1), 1990.

[Pag88] Meilir Page-Jones. *The Practical Guide to Structured Systems Design.* Yourdon Press, 1988.

[Pet81] J. L. Peterson. *Petri-Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[PS83] H. Partsch and R. Steinbruggen. Program transformation systems. *Computing Surveys*, 15(3):199–236, September 1983.

[Rom90] H. Dieter Rombach. Design measurement: Some lessons learned. *IEEE Software*, pages 17–25, March 1990.

[RS89] C. Riesbeck and R. Schank. *Inside Case-Based Reasoning.* Hillsdale, 1989.

[Sac92] Krzystof M. Sacha. Transformational implementation of PAISLey specifications using Petri nets. *Software Engineering Journal*, 7(3):191–204, May 1992.

[Sha87] H. Sharp. Accelerating the analysis/design transition with the transform analyser. In *Conference on Automatic System Development, Research Papers*, pages 1–15. Leicester Polytechnique, April 1987.

[Sho88] Peretz Shoval. ADISSA: Architectural design of information systems based on structured analysis. *Information Systems*, 13(2):193–210, 1988.

[SM90] P. Shoval and O. Manor. Software engineering tools supporting ADISSA methodology for systems analysis and design. *Information and Software Technology*, 32(5):357–369, June 1990.

[STM88] Paul G. Sorenson, Jean-Paul Tremblay, and Andrew J. McAllister. The metaview system for many specification environments. *IEEE Software*, 5(2):30–38, March 1988.

[TK91] Yonglei Tao and Chenho Kung. Formal definition and verification of data flow diagrams. *Journal of Systems and Software*, 16(1):29–36, Sept 1991.

[TR88] J. Tsai and J. Ridge. Intelligent support for specifications transformation. *IEEE Software*, 5:28–35, November 1988.

[TVJ92] Graham Tate, June Verner, and Ross Jeffery. CASE: A testbed for modeling, measurement and management. *Communications of the ACM*, 35(4):65–72, April 1992.

[War86] Paul T. Ward. The transformation schema: An extension of the data flow diagram to represent control and timing. *IEEE Transactions on Software Engineering*, 12(2):198–210, February 1986.

[YC78] E. Yourdon and L. Constantine. *Structured Design.* Yourdon Press, 1978.

[Zav82] P. Zave. An operational approach to requirements specification for embedded systems. *IEEE Transactions on Software Engineering*, 8(5):250–269, 1982.

[ZS86] P. Zave and W. Schell. Salient features of an executable specification language and its environment. *IEEE Transactions on Software Engineering*, 12(2):312–315, 1986.

Appendix A

BNF Description of the Proposed ETL

An extended BNF description is used to define the syntax of the proposed ETL. We adopt the following standard set of metanotions. Enclosing an expression within brace brackets ('{ ... }') indicates that zero or more occurrences of that expression may appear in the language. Square brackets ('[...]') indicate zero or one occurrence of an expression. A symbol enclosed in quotes (as in ';') indicates that the symbol is part of the ETL. The symbol ' | ' is used to indicate a choice of expressions. Reserved words are capitalized. All terminal words are typeset using typewriter font (as in ⟨**rule name**⟩) and are treated as ⟨**identifier**⟩ in the grammer. Comments are parenthesized with /* ... */.

/* Skeleton */
⟨etl source⟩ ::= ⟨trans header⟩
⟨env header⟩
{ ⟨declaration⟩ ';' }
⟨rule set⟩
⟨rules⟩
⟨trans header⟩ ::= TRANSFORMATION ⟨**identifier**⟩ ';'
⟨env header⟩ ::= ENVIRONMENT ⟨**env name**⟩ → ⟨**env name**⟩ ';'
⟨rule set⟩ ::= RULE SET ⟨rule invocations⟩
⟨rule invocations⟩ ::= ⟨rule invocation⟩ ';' { ⟨rule invocations⟩ }
⟨rules⟩ ::= ⟨rule⟩ { ⟨rules⟩ }
/* Declaration syntax */
⟨declaration⟩ ::= ⟨data type⟩ ':' ⟨**identifier**⟩ [← ⟨rvalue⟩]
/* Data type */
⟨data type⟩ ::= ⟨fundamental data type⟩ | ⟨array type⟩ | ⟨set type⟩
| ⟨base type⟩ | ⟨trans attr type⟩ | ⟨trans agg type⟩
⟨fundamental data type⟩ ::= BOOLEAN | INTEGER | STRING
⟨base type⟩ ::= ⟨**entity name**⟩ | ⟨**relationship name**⟩ | ⟨**aggregate name**⟩
⟨array type⟩ ::= ARRAY OF ⟨data type⟩
⟨set type⟩ ::= SET OF ⟨data type⟩
/* Transformation attribute */
⟨trans attr type⟩ ::= TRANS ATTR ⟨fundamental data type⟩ OF ⟨base type⟩
/* Transformation aggregate */
⟨trans agg type⟩ ::= TRANS AGGREGATE OF ⟨**aggregate name**⟩

```
/* Arithmetic, boolean, and set expression */
⟨rvalue⟩ ::=   ⟨lvalue⟩
            | ⟨integer constant⟩
            | '⟨string constant⟩'
            | ⟨boolean constant⟩
            | ⟨relationship call⟩
            | ⟨relationship variable⟩'.'⟨role name⟩
            | ENDOF '('⟨array variable⟩ ')'
            | EMPTY_SET
            | ⟨image set⟩
            | SIZE_OF '('⟨any set⟩ ')'
            | LOCATE_AGGREGATE '('trans agg variable ')'
⟨lvalue⟩ ::=  ⟨integer variable⟩
            | ⟨string variable⟩
            | ⟨boolean variable⟩
            | ⟨base variable⟩
            | ⟨set variable⟩
            | ⟨trans agg variable⟩
            | ⟨array ref⟩
            | ⟨trans attr ref⟩
            | ⟨base attr ref⟩
⟨relationship call⟩ ::= ⟨relationship name⟩ '(' ⟨rel arg⟩ { ',' ⟨rel arg⟩ } ')'
⟨rel arg⟩ ::= '*' | '?' | ⟨base variable⟩
⟨image set⟩ ::= MAP_OF '(' ⟨entity name⟩ ')' WITHIN ⟨aggregate variable⟩
⟨any set⟩ ::= ⟨dynamic set⟩ | ⟨static set⟩
⟨dynamic set⟩ ::= ⟨image set⟩ | ⟨set variable⟩
⟨static set⟩ ::= ⟨set type⟩ ⟨within clause⟩ [ WHERE ⟨boolean exp⟩ ]
⟨within clause⟩ ::=   WITHIN ⟨aggregate name⟩
                    | WITHIN ⟨tran agg variable⟩
⟨base variable⟩ ::= ⟨component variable⟩ | ⟨aggregate variable⟩
⟨component variable⟩ ::= ⟨entity variable⟩ | ⟨relationship variable⟩
⟨array ref⟩ ::= ⟨array variable⟩ '[' ⟨integer value⟩ ']'
⟨integer value⟩ ::= ⟨integer constant⟩ | ⟨integer variable⟩
⟨trans attr ref⟩ ::= ⟨base variable⟩'.'⟨identifier⟩
⟨base attr ref⟩ ::= ⟨base variable⟩'.'⟨attribute name⟩
⟨boolean constant⟩ ::= TRUE | FALSE
⟨exp⟩ ::= ⟨rvalue⟩ | ⟨arith or set exp⟩ | '(' ⟨exp⟩ ')'
⟨boolean exp⟩ ::=   ⟨exp⟩⟨rel op⟩⟨exp⟩
                  | ⟨relationship call⟩
                  | TYPE_OF '(' ⟨base variable⟩ ')' '=' ⟨base type⟩
                  | ⟨base variable⟩ '=' NULL
                  | ⟨base variable⟩ IN ⟨any set⟩
                  | IS_EMPTY '(' ⟨any set⟩ ')' [ '=' ⟨boolean constant⟩ ]
                  | NOT ⟨boolean exp⟩
                  | ⟨boolean exp⟩⟨boolean op⟩⟨boolean exp⟩
                  | '(' ⟨boolean exp⟩ ')'
⟨rel op⟩ ::= '=' | '!=' | '<=' | '>=' | '>' | '<'
⟨arith or set exp⟩ ::= ⟨exp⟩⟨math op⟩⟨exp⟩ | ⟨exp⟩⟨set op⟩⟨exp⟩
⟨math op⟩ ::= '+' | '-' | '*' | '/'
⟨set op⟩ ::= UNION | INTERSECT
⟨boolean op⟩ ::= AND | OR
/* Rule format */
⟨rule invocation⟩ ::= ⟨rule name⟩ [ '(' ⟨rvalue list⟩ ')' ]
```

⟨rvalue list⟩ ::= ⟨rvalue⟩ { ',' ⟨rvalue⟩ }
⟨rule⟩ ::= ⟨rule header⟩
{ ⟨declaration⟩';' }
{ ⟨stmt⟩';' }
⟨rule header⟩ ::= RULE ⟨**rule name**⟩ ['(' ⟨parm decls⟩ ')']';'
⟨parm decls⟩ ::= ⟨parm decl⟩ { ',' ⟨parm decls⟩ }
⟨parm decl⟩ ::= ⟨**identifier**⟩ ':' ⟨data type⟩
/* Statements */
⟨stmt⟩ ::= ⟨assignment stmt⟩
| ⟨create stmt⟩
| ⟨destroy stmt⟩
| ⟨insert stmt⟩
| ⟨delete stmt⟩
| ⟨oracle input stmt⟩
| ⟨oracle output stmt⟩
| ⟨rule invocation⟩
| EXTERNAL ⟨rule invocation⟩
| ⟨find stmt⟩
| ⟨for clause⟩ { ⟨stmt⟩';' } ENDFOR
| ⟨if clause⟩ { ⟨stmt⟩';' } ENDIF
| ⟨if clause⟩ { ⟨stmt⟩';' } ELSE { ⟨stmt⟩';' } ENDIF
⟨assignment stmt⟩ ::= ⟨lvalue⟩ ← ⟨exp⟩
⟨create stmt⟩ ::= CREATE ENTITY ⟨**entity name**⟩ '(' ⟨**identifier**⟩ ')'
WITHIN ⟨**aggregate variable**⟩
| CREATE RELATIONSHIP ⟨**relationship name**⟩ '(' ⟨rvalue list⟩ ')'
[⟨attribute assignment⟩] WITHIN ⟨**aggregate variable**⟩
| CREATE AGGREGATE ⟨**aggregate name**⟩ '(' ⟨**identifier**⟩ ')'
| CREATE TRANS AGGREGATE ⟨**aggregate name**⟩ '(' ⟨**tran agg variable**⟩ ')'
WITHIN ⟨**aggregate variable**⟩
⟨attribute assignments⟩ ::= '(' ⟨**attribute name**⟩ ← ⟨exp⟩ { ';' ⟨**attribute name**⟩ ← ⟨exp⟩ } ')'
⟨destroy stmt⟩ ::= DESTROY ENTITY ⟨**entity name**⟩ '(' ⟨**identifier**⟩ ')'
WITHIN ⟨ **aggregate variable**⟩
| DESTROY RELATIONSHIP ⟨**relationship name**⟩ '(' ⟨rvalue list⟩ ')'
WITHIN ⟨**aggregate variable**⟩
| DESTROY AGGREGATE ⟨**aggregate name**⟩ '(' ⟨**identifier**⟩ ')'
| DESTROY TRANS AGGREGATE ⟨**aggregate name**⟩ '(' ⟨**tran agg variable**⟩ ')'
WITHIN ⟨**aggregate variable**⟩
⟨insert stmt⟩ ::= INSERT '(' ⟨component variable⟩ ',' ⟨**trans agg variable**⟩ ')'
| INSERT '(' ⟨**relationship name**⟩ '(' ⟨rvalue list⟩ ')' ',' ⟨**trans agg variable**⟩ ')'
| INSERT '(' ⟨**entity variable**⟩ ',' ⟨image set⟩ ')'
| INSERT '(' ⟨base variable⟩ ',' ⟨**set variable**⟩ ')'
| INSERT '(' ⟨**relationship name**⟩ '(' ⟨rvalue list⟩ ')' ',' ⟨**set variable**⟩ ')'
⟨delete stmt⟩ ::= DELETE '(' ⟨component variable⟩ ',' ⟨**trans agg variable**⟩ ')'
| DELETE '(' ⟨**relationship name**⟩ '(' ⟨rvalue list⟩ ')' ',' ⟨**trans agg variable**⟩ ')'
| DELETE '(' ⟨**entity variable**⟩ ',' ⟨image set⟩ ')'
| DELETE '(' ⟨base variable⟩ ',' ⟨**set variable**⟩ ')'
| DELETE '(' ⟨**relationship name**⟩ '(' ⟨rvalue list⟩ ')' ',' ⟨**set variable**⟩ ')'
⟨oracle input stmt⟩ ::= ORACLE INPUT ⟨lvalue⟩
⟨oracle output stmt⟩ ::= ORACLE OUTPUT ⟨rvalue list⟩
⟨for clause⟩ ::= FOR ALWAYS
| FOR EACH ⟨lvalue⟩ IN ⟨any set⟩
| FOR ⟨**integer variable**⟩ ← ⟨rvalue⟩ TO ⟨rvalue⟩
⟨if clause⟩ ::= IF ⟨boolean exp⟩

⟨find stmt⟩ ::= FIND ⟨lvalue⟩ IN ⟨any set⟩

Appendix B

ETL Specifications

In this appendix the parts of the ETL specification of the ADISSA transformation that are scattered throughout the thesis are presented together.

B.1 ETL Specification of the ADISSA Transformation

```
TRANSFORMATION adissa_transformation;
  *** ETL specification of the ADISSA transformation
  ENVIRONMENT dfd -> fsm;
  *** The source environment is the DFD environment specification and
  *** the target environment is the FSM environment specification.

  ***
  *** Transformation attributes
  ***
  *** interconnect is used to partition the set of all processes.
  TRANS ATTR BOOLEAN OF process: interconnect <- FALSE;
  *** The names of the coming three transformation attributes have
  *** explained their purposes.
  TRANS ATTR BOOLEAN OF terminator: belong_to_starting_set;
  TRANS ATTR BOOLEAN OF terminator: belong_to_ending_set;
  TRANS ATTR BOOLEAN OF dataStore: belong_to_ending_set;
  *** If cdf is set to TRUE, the corresponding data flow is a CDF.
  TRANS ATTR BOOLEAN OF dataFlow: cdf <- FALSE;
  *** IF involve is set to TRUE, the corresponding data flow is involved
  *** in an AND-relationship.
  TRANS ATTR BOOLEAN OF dataFlow: involve <- FALSE;
  *** Every CDF that is involved in an AND-relationship is assigned unique
  *** integer bigger than 0.
  TRANS ATTR INTEGER OF dataFlow: p_num <- 0;
  *** Indicate which associated_tree relationship instances are removed
  *** from the database state.
  TRANS ATTR BOOLEAN OF tree: mark <- FALSE;
  ***
  *** Transformation Aggregates
  ***
  *** Array of transformation aggregates to capture the
  *** ADISSA-transactions.
  ARRAY OF TRANS AGGREGATE OF data_flow_aggregate: adissa_trans;

*** Sequences of rules to be initiated.
```

```
RULE SET
  *** First we partition the set of processes into components.
  identify_adissa_transaction;
  *** Then, we start the ADISSA transformation.
  begin_transformation;

*** The following rule partitions the set of processes.
RULE identify_adissa_transaction(agg: data_flow_aggregate);
  process: p, q;
  INTEGER: i;
  i <- 0;
  FOR EACH p IN SET OF process WITHIN agg WHERE p.interconnect = FALSE
    i <- i+1;
    CREATE TRANS AGGREGATE (adissa_trans[i]) WITHIN agg;
    INSERT(p,adissa_trans[i]);
    p.interconnect <- TRUE;
    find_interconnect_process(p,adissa_trans[i]);
  ENDFOR;

*** The following rule finds the component that contains p and put
*** the component into adissa.
RULE find_interconnect_process(p: process, adissa: TRANS AGGREGATE OF
                                                   data_flow_aggregate);
  process: q;
  data_flow_aggregate: agg;

  agg <- LOCATE_AGGREGATE(adissa);
  *** Retrieve the original aggregate.
  FOR EACH q IN SET OF process WITHIN agg WHERE (sends(p,*,q)
  OR sends(q,*,p)) AND q.interconnect = FALSE
    q.interconnect <- TRUE;
    INSERT(q,adissa);
    find_interconnect_process(q,adissa);
  ENDFOR;

*** Begin the ADISSA transformation.
RULE begin_transformation;
  INTEGER: i;

  *** Each ADISSA transaction is transformed into the target environment
  *** sequentially.
  FOR i <- 1 TO ENDOF(adissa_trans)
    *** Reset the transformation attributes.
    reset_trans_attributes(adissa_trans[i]);
    *** Construct the ADISSA-transaction.
    create_adissa_transaction(adissa_trans[i]);
    *** Apply the SEC metrics before we proceed.
    apply_sec_metric(adissa_trans[i]);
    *** Construct the FSM-transaction.
    create_fsm_transaction(adissa_trans[i]);
    *** Transform the FSM-transaction into FSM(s) in the target environment.
    analyze_fsm_transaction(adissa_trans[i]);
  ENDFOR;

*** A process can belong to one and only one ADISSA-transaction because of
*** the partitioning of the set of all elementary processes. The same
*** applies to data flows because by assumption no data flow links more
```

```
*** than two objects. Only a terminator or a data store can belong to
*** several ADISSA-transactions. Each ADISSA-transaction is transformed
*** into the target environment sequentially and all the ADISSA-transaction
*** are represented as transformation aggregates which can intersect with
*** each other. Each time before an ADISSA-transaction is transformed, we
*** must reset the transformation attributes attached to the entities
*** terminator and dataStore to the initial values.
RULE reset_trans_attributes(adissa: TRANS AGGREGATE OF data_flow_aggregate);
  terminator: t;
  dataStore: ds;

  FOR EACH t IN SET OF terminator WITHIN adissa
    t.belong_to_starting_set <- FALSE;
    t.belong_to_ending_set <- FALSE;
  ENDFOR;

  FOR EACH ds IN SET OF dataStore WITHIN adissa
    ds.belong_to_ending_set <- FALSE;
  ENDFOR;

*** The transformation aggregate "adissa" contains a set of
*** processes which are weakly connected.
*** We extend adissa to include all other elements plus the relationships
*** that are linked to the set of processes. If adissa does not contain
*** any terminator, we ask the user to supply one. It is necessary
*** because every ADISSA-transaction must have a starting set which
*** consists of terminator(s). Unfortunately such a condition cannot be
*** checked by ECL.
RULE create_adissa_transaction(adissa: TRANS AGGREGATE OF data_flow_aggregate);
  sends: s;
  changes: ch;
  reads: r;
  terminator: t;
  process: q;
  dataFlow: df;
  data_flow_aggregate: agg;
  STRING: buf;

  agg <- LOCATE_AGGREGATE(adissa);
  *** Since inserting a relationship instance to a transformation
  *** aggregate automatically inserts its entity participants to the
  *** transformation aggregate, we can simply the process of constructing
  *** ADISSA-transaction by just examining and inserting the sends, reads
  *** and changes relationship instances into the appropriate
  *** transformation aggregates.
  FOR EACH s IN SET OF sends WITHIN agg WHERE (s.source IN adissa) OR
  (s.destination IN adissa)
    INSERT(s,adissa);
  END;

  FOR EACH ch IN SET OF changes WITHIN agg WHERE (ch.name IN adissa)
    INSERT(ch,adissa);
  END;

  FOR EACH r IN SET OF reads WITHIN agg WHERE (r.name IN adissa)
    INSERT(r,adissa);
  END;
```

```
    *** Make sure the ADISSA transaction contains at least one terminator.
    FIND t IN SET OF terminator WITHIN adissa;
    IF t = NULL
      ORACLE OUTPUT "The ADISSA transaction contains ";
      ORACLE OUTPUT "no starting terminator";
      ORACLE OUTPUT "Please supply the name of the terminator which";
      ORACLE OUTPUT "activates the transaction";
      EXTERNAL display_trans_aggregate(adissa);
      ORACLE INPUT buf;
      FIND t IN SET OF terminator WITHIN agg WHERE t.name = buf;
      *** For convenience, we do not check if we have found the object
      *** successfully. We assume we have.
      ORACLE OUTPUT "What process does the terminator trigger?"
      ORACLE INPUT buf;
      FIND q IN SET OF process WITHIN agg WHERE q.name = buf;
      *** For the same reason, we assume the user has entered the correct
      *** name for the process.
      *** We must create an extra data flow to link the terminator and
      *** the process because if there were such a data flow, we would
      *** have found it already.
      CREATE ENTITY dataFlow (df) WITHIN agg;
      ORACLE OUTPUT "Please supply a name for the data flow which links"
      ORACLE OUTPUT t.name, " and ", q.name;
      ORACLE INPUT df.name;
      df.involve <- FALSE;
      CREATE RELATIONSHIP sends(t,df,q) WITHIN agg;
      INSERT(sends(t,df,q),adissa);
    ENDIF;

  *** The SEC metrics is applied at this stage. Designer can take
  *** corrective actions before proceeding with the transformation.
  *** We examine each data flow that links two processes in the
  *** ADISSA-transaction.
  RULE apply_sec_metric(adissa: TRANS AGGREGATE OF data_flow_aggregate);
    dataFlow df;
    sends: s;

    FOR EACH s IN SET OF sends WITHIN adissa WHERE
    TYPE_OF(s.source) = process AND TYPE_OF(s.destination) = process
      df <- s.data;
      sec_metric(df,adissa);
    ENDFOR;

  *** The actual SEC metrics is performed by the following rule.
  RULE sec_metric(df: dataFlow, adissa: TRANS AGGREGATE OF data_flow_aggregate);
    TRANS AGGREGATE OF data_flow_aggregate: orig;
    data_flow_aggregate: agg;
    process p, q, f;
    sends: s;
    SET OF process: so, de;
    dataFlow: dff;
    INTEGER: k, m, n;

    agg <- LOCATE_AGGREGATE(adissa);
    CREATE TRANS AGGREGATE (orig) WITHIN agg;
    *** Work on the copy ***
```

```
  orig <- adissa;
  *** Calculate the total number of processes.
  k <- 0;
  FOR EACH p IN SET OF process WITHIN orig
    k <- k+1;
  ENDFOR;

  *** Locate the initial and terminal point of the data flow df.
  FIND s IN SET OF sends WITHIN orig WHERE s.data = df;
  p <- s.source;
  q <- s.destination;

  *** remove the instance df and any other relationship instances that
  *** contain df from the transformation aggregate "orig".
  DELETE(df,orig);
  *** reset the transformation attribute "interconnect" for the
  *** transformation aggregate "orig".
  FOR EACH dff IN SET OF dataFlow WITHIN orig
    dff.interconnect <- FALSE;
  ENDFOR;
  *** Find the two sets of processes that contain p and q respectively.
  find_connect_process(p,orig,so);
  find_connect_process(q,orig,de);

  m <- SIZE_OF(so);
  n <- SIZE_OF(de);
  ORACLE OUTPUT "Data flow ", df.name, "has the following SEC metric value:";
  ORACLE OUTPUT k,m,n;

*** The following rule is similar to the rule find_interconnect process.
RULE find_connect_process(p: process, orig: TRANS AGGREGATE OF
                          data_flow_aggregate, s: SET OF process);
  process: q;
  FOR EACH q IN SET OF process WITHIN orig WHERE (sends(p,*,q)
  OR sends(q,*,p)) AND q.interconnect = FALSE
    q.interconnect <- TRUE;
    INSERT(q,s);
    find_connect_process(q,orig,s);
  ENDFOR;

*** Initiate the rules to convert an ADISSA-transaction into a
*** FSM-transaction. Each interpretation is stored as a transformation
*** aggregate.
RULE create_fsm_transaction(adissa: TRANS AGGREGATE OF data_flow_aggregate);
  INTEGER: num, i;
  *** Each interpretation (subtask) is stored as a transformation
  *** aggregate.
  ARRAY OF TRANS AGGREGATE OF data_flow_aggregate: itpt;
  data_flow_aggregate: dfdagg;

  dfdagg <- LOCATE_AGGREGATE(adissa);
  ORACLE OUTPUT "How many interpretations are there for the ADISSA ";
  ORACLE OUTPUT "transaction?"
  *** Display the ADISSA-transaction to the user.
  EXTERNAL display_trans_aggregate(adissa);
  *** Ask for input.
  ORACLE INPUT num;
```

```
    *** Get all the interpretations.
    FOR i <- 1 TO num
      CREATE TRANS AGGREGATE (itpt[i]) WITHIN dfdagg;
      ORACLE OUTPUT "Please specify components for the ";
      ORACLE OUTPUT i, " interpretation";
      *** The routine prompt_for_interpretation will do the following:
      ***   (1) The ADISSA-transaction adissa will be shown to the user
      ***       perhaps graphically.
      ***   (2) The user will select the entities and relationships
      ***       that will participate in the interpretation.
      ***   (3) The selected elements will be put in the transformation
      ***       aggregate v[i].
      EXTERNAL prompt_for_interpretation(adissa,itpt[i]);
      *** Apply the rules to formalize the process logic of each
      *** interpretation.
      specify_starting_set(itpt[i]);
      specify_ending_set(itpt[i]);
      specify_boolean_expression(itpt[i]);
      specify_cdf_and_df(itpt[i]);
      add_additional_cdf(itpt[i]);
    ENDFOR;
    *** Finally merge all the interpretation to form the FSM-transaction.
    merge_interpretations(adissa,itpt);

*** Ask the user to specify the starting set.
RULE specify_starting_set(interpretation: TRANS AGGREGATE OF
                                          data_flow_aggregate)
    terminator: t;
    sends: s;
    STRING: buf;

    FOR EACH t IN SET OF terminator WITHIN interpretation
      *** Only terminator with at least one outgoing flow can be in
      *** the starting set.
      FIND s IN SET OF sends WITHIN interpretation WHERE s.source = t;
      IF NOT s = NULL
        *** t has at least one outgoing flow ***
        ORACLE OUTPUT "Does the terminator ";
        ORACLE OUTOUT t.name;
        ORACLE OUTPUT "belong to the starting set?";
        ORACLE INPUT buf;
        IF buf = "yes"
          t.belong_to_starting_set <- TRUE;
        ENDIF;
      ENDIF;
    ENDFOR;

*** Ask the user to specify the ending set.
RULE specify_ending_set(interpretation: TRANS AGGREGATE OF
                                        data_flow_aggregate);
    terminator: t;
    dataStore: ds;
    sends: s;
    STRING: buf;

    FOR EACH t IN SET OF terminator WITHIN interpretation
      *** Only terminator with at least one incoming flow can be in
```

```
    *** the ending set.
    FIND s IN SET OF sends WITHIN interpretation WHERE s.destination = t;
    IF NOT s = NULL
      *** t has at least one incoming flow ***
      ORACLE OUTPUT "Does the terminator";
      ORACLE OUTPUT t.name;
      ORACLE OUTPUT "belong to the ending set";
      ORACLE INPUT buf;
      IF buf = "yes"
        t.belong_to_ending_set <- TRUE;
      ENDIF;
    ENDIF;
  ENDFOR;

  FOR EACH ds IN SET OF dataStore WITHIN interpretation
    ORACLE OUTPUT "Does the data store";
    ORACLE OUTPUT t.name;
    ORACLE OUTPUT "belong to the ending set";
    ORACLE INPUT buf;
    IF buf = "yes"
      ds.belong_to_ending_set <- TRUE;
    ENDIF;
  ENDFOR;

*** Specify the boolean expression
RULE specify_boolean_expression(interpretation:
                                TRANS AGGREGATE OF data_flow_aggregate);
  INTEGER: j;
  STRING: buf;
  process: p;
  dataFlow: df;
  SET OF dataFlow: b;
  tree: top;
  data_flow_aggregate: agg;

  agg <- LOCATE_AGGREGATE(interpretation);
  FOR EACH p IN SET OF process WITHIN interpretation
    j <- 0;
    b <- EMPTY_SET;
    *** By assumption each data flow participates
    *** in only one relationship.
    *** We count how many outgoing data flows from p by counting
    *** how many relationships that contain p and df.
    FOR EACH df IN SET OF dataFlow WITHIN interpretation
      IF sends(p,df,*)
        INSERT(df,b);
        j <- j+1;
      ENDIF;
      IF changes(p,df,*)
        INSERT(df,b);
        j <- j+1;
      ENDIF;
    ENDFOR;
    *** If the process has more than one outgoing flow, ask the user
    *** to specify a boolean expression for the data flow.
    IF j > 1
      ORACLE OUTPUT "Please specify a boolean relation among the flows"
```

```
      FOR EACH df IN b
        ORACLE OUTPUT df.name;
      ENDFOR;
      ORACLE INPUT buf;
      *** Create a tree instance in preparation for the relationship
      *** associated_tree.
      CREATE ENTITY tree (top) WITHIN agg;
      *** Assign a unique name to top.
      EXTERNAL assign_unique_name(top,agg);
      *** The entity "top" marks the top tree.
      CREATE RELATIONSHIP associated_tree(p,top) WITHIN agg;
      INSERT(associated_tree(p,top),interpretation);
      *** We need an external function which takes a logical expression
      *** stored in buf and construct a tree starting at top and put
      *** the whole tree in the transformation aggregate interpretation.
      EXTERNAL create_tree(interpretation,top,buf);
    ENDIF;
  ENDFOR;

*** Specify CDF in this rule.
RULE specify_cdf_and_df(interpretation: TRANS AGGREGATE OF
                                        data_flow_aggregate);
  process: p;
  dataFlow: df;
  dataStore: ds;
  SET OF dataFlow: b;
  tree: top;
  changes: ch;
  sends: s;
  STRING: answer;

  *** According to Babin's paper, we do the following:
  *** Any flow between two processes, or a process and a terminator is
  *** converted to a CDF.
  *** If a flow is from a process to a data store, then
  *** (a) if it is not involved in an AND, it is converted to a CDF
  *** (b) if it is involved in an AND with flows going only to data
  ***     stores, it is converted to a CDF.
  *** Finally, we make sure ANDs involve only CDFs.

  *** Convert any data flow participants in any sends relationship to
  *** a CDF.
  FOR EACH s IN SET OF sends WITHIN interpretation
    df <- s.data;
    df.cdf <- TRUE;
  END;

  *** Look for those processes that participate in changes relationship
  *** with more than one outgoing flow.
  FOR EACH p IN SET OF process WITHIN interpretation
    b <- EMPTY_SET;
    FOR EACH ch IN SET OF changes WITHIN interpretation WHERE ch.name = p
      df <- ch.flow;
      set_entry_point(df,interpretation);
      *** After returning from the rule, if a CDF is involved in an
      *** AND-relationship, the involve attribute is set to TRUE.
      IF df.involve = TRUE
```

```
      INSERT(df,b);
    ELSE
      *** The data flow is not involved in an AND,
      *** it is thus converted
      *** to a CDF. Moreover, the data store should belong to the
      *** ending set.
      df.cdf <- TRUE;
      ds.belong_to_ending_set <- TRUE;
    ENDIF;
  ENDFOR;
  IF SIZE_OF(b) > 1
    *** We need an external function which takes the top node and a
    *** set of dataFlows and determines how the data flows
    *** in the given set are involved in AND and hence changes
    *** those dataflows to CDFs accordingly.
    *** For example, if we have the expression
    *** (d1 AND d2) XOR (d3 AND d4) where d1 is outgoing to
    *** a process, and d2, d3 and d4 are outgoing to data
    *** stores. By construction, set b will contain all
    *** d1, d2, d3 and d4. According to the
    *** Babin"s rule, only d1, d3 and d4 are changed to CDF.
    top <- associated_tree(p,?);
    EXTERNAL analyze_and_relation(top,b,interpretation);
    *** Re-determine whether a data store should
    *** belong to the ending set.
    FOR EACH ch IN SET OF changes WITHIN interpretation WHERE ch.flow IN b
    AND ch.flow.cdf = TRUE;
      ch.data.belong_to_ending_set <- TRUE;
    ENDFOR;
    *** We need an external function which takes a tree
    *** and modify it so that ANDs involve only CDFs.
    *** The external function also makes sure the
    *** transformation attribute "involve" of a dataFlow
    *** is set to true if and only if it is involved in an
    *** AND. The "involve" attribute has been set before.
    *** It is to make sure after modifying the AND
    *** relationship, the "involve" attribute
    *** should reflect the changes.
    EXTERNAL and_involve_only_cdf(top,interpretation);
  ENDIF;
ENDFOR;

*** Specify additional CDF if the flow of control is in the opposite
*** direction of the data flow.
RULE add_additional_cdf(interpretation: TRANS AGGREGATE OF
                                        data_flow_aggregate);
  sends: s;
  dataFlow: df;
  data_flow_aggregate: agg;
  STRING: answer;

  agg <- LOCATE_AGGREGATE(interpretation);
  FOR EACH s IN SET OF sends WITHIN interpretation
    *** Ask the user to specify any additional CDF.
    ORACLE OUTPUT "For the sends relationship with"
    ORACLE OUTPUT "source:", s.source.name;
    ORACLE OUTPUT "data:", s.data.name;
```

```
    ORACLE OUTPUT "destination:", s.destination.name;
    ORACLE OUTPUT "Is there any CDF going in the opposite "
    ORACLE OUTPUT "direction? (YES/NO)";
    ORACLE INPUT answer;
    IF answer = "YES"
      CREATE ENTITY dataFlow (df) WITHIN agg;
      ORACLE OUTPUT "Please give a name for the new CDF";
      ORACLE INPUT df.name;
      df.cdf <- TRUE;
      df.involve <- FALSE;
      CREATE RELATIONSHIP sends(s.destination, df, s.source) WITHIN agg;
      INSERT(sends(s.destination,df,s.source),interpretation);
      IF s.source IN SET OF terminator WITHIN interpretation
        s.source.belong_to_ending_set <- TRUE;
      ENDIF;
      IF s.destination IN SET OF terminator WITHIN interpretation
        s.destination.belong_to_starting_set <- TRUE;
      ENDIF;
    ENDIF;
  ENDFOR;

*** The following rule determine whether a data flow is involved in
*** an AND-relationship.
RULE set_entry_point(df: dataFlow, vg: TRANS AGGREGATE OF
                                       data_flow_aggregate);
  *** Set the attribute involve of a data flow to be
  *** TRUE if that data flow is involved in an AND.
  connective: con;

  FIND con IN SET OF connective WITHIN vg WHERE con.left_tree = df
  OR con.right_tree = df
  IF NOT con = NULL
    *** trace up the binary tree.
    trace_parent(df,con,vg);
  ENDIF;

RULE trace_parent(df: dataFlow, con: connective,
                  vg: TRANS AGGREGATE OF data_flow_aggregate);
  tree: root;
  connective: rel;

  IF con.type = "and_type"
    df.involve <- TRUE;
  ELSE
    root <- con.root;
    FIND rel IN SET OF connective WITHIN vg WHERE rel.left_tree = root
    OR rel.right_tree = root;
    IF NOT rel = NULL
      trace_parent(df,rel,agg);
    ENDIF;
  ENDIF;

*** Merge all the interpretations to form an FSM-transaction.
RULE merge_interpretations(adissa: TRANS AGGREGATE OF data_flow_aggregate,
                           m: ARRAY OF TRANS AGGREGATE OF
                              data_flow_aggregate);
  INTEGER: i;
```

```
  *** Merging is as simple as taking the union of all the
  *** interpretations with the parent.
  FOR i <- 1 TO ENDOF(m)
    adissa <- adissa UNION m[i];
  ENDFOR;
  *** Construct a boolean expression from all the expressions specified
  *** in each interpretation.
  merge_boolean_expressions(adissa,m);
  *** Merge the terminator.
  merge_terminators(adissa);

*** Join all the logical expressions in the interpretations using XOR.
RULE merge_boolean_expressions(adissa: TRANS AGGREGATE OF data_flow_aggregate,
                               m: ARRAY OF TRANS AGGREGATE OF
                                  data_flow_aggregate);
  data_flow_aggregate: fsmtran;
  tree: top, root, apex;
  ARRAY OF tree: s;
  associated_tree: as;
  process: p;
  dataFlow: df;
  ARRAY OF dataFlow: adf;
  sends: sd;
  changes: ch;
  INTEGER: c, numt, i, numdf;

  *** Make sure we have more than one interpretations to merge.
  fsmtran <- LOCATE_AGGREGATE(m[1]);
  IF ENDOF(m) > 1
    FOR EACH p IN SET OF process WITHIN adissa
      *** Find out how many outgoing control data flows from p.
      c <- 0;
      FOR EACH sd IN SET OF sends WITHIN adissa
      WHERE sd.source = p AND sd.data.cdf = TRUE
        c <- c+1;
      ENDFOR;
      FOR EACH ch IN SET OF changes WITHIN adissa
      WHERE ch.name = p AND ch.flow.cdf = TRUE
        c <- c+1;
      ENDFOR;
      *** When p has more than one outgoing flows,
      *** we must create an explicit relation among
      *** the flows. Also for all groups of flows
      *** outgoing from a process, an algebraic expression
      *** composed of the expression of each interpretation,
      *** separated by XOR, is created.
      IF c > 1
        numt <- 0;
        numdf <- 0;
        FOR i <- 1 TO ENDOF(m)
          FIND as IN SET OF associated_tree WITHIN m[i] WHERE as.pname = p;
          IF as = NULL
            *** For each interpretation, put any data flow not
            *** appear in any tree logical structure in an
            *** array.
            FOR EACH df IN SET OF dataFlows WITHIN m[i]
            WHERE (sends(p,df,*) OR changes(p,df,*))
```

```
    AND df.cdf = TRUE
      numdf <- numdf+1;
      adf[numdf] <- df;
    ENDFOR;
  ELSE
    *** We find an associated tree.
    numt <- numt+1;
    s[numt] <- as(p,?);
  ENDIF;
ENDFOR;
IF numt > 1
  *** Join all the trees found so far
  *** if there are more than one.
  merge_tree(p,s,numt,adissa);
ELSE IF numt = 1
  *** If there is only one tree, mark this tree.
  s[1].mark <- TRUE;
ENDIF;
IF numdf > 1
*** There are more than one single cdf, join them first.
  CREATE ENTITY tree (root) WITHIN fsmtran;
  INSERT(root,adissa);
  EXTERNAL assign_unique_name(root,fsmtran);
  CREATE RELATIONSHIP connective(root,adf[1],adf[2])
  (type <- "xor_type") WITHIN fsmtran;
  INSERT (connective(root,adf[1],adf[2]), adissa);
  FOR i <- 3 TO numdf
    top <- root;
    CREATE ENTITY tree (root) WITHIN fsmtran;
    INSERT(root, adissa);
    EXTERNAL assign_unique_name(root,fsmtran);
    CREATE RELATIONSHIP connective(root,top,adf[i])
    (type <- "xor_type") WITHIN fsmtran;
    INSERT(connective(root,top,adf[i], adissa);
  ENDFOR;
  IF numt > 0
    *** Join the previous tree.
    FIND as IN SET OF associated_tree
    WITHIN fsmtran WHERE as(p,*) AND as.top_root.mark = TRUE;
    top <- as(p,?);
    top.mark <- FALSE;
    CREATE ENTITY tree (apex) WITHIN fsmtran;
    INSERT(apex,adissa);
    EXTERNAL assign_unique_name(apex,fsmtran);
    CREATE RELATIONSHIP connective(apex,root,top)
    (type <- "xor_type") WITHIN fsmtran;
    INSERT(connective(apex,root,top), adissa);
    apex.mark <- TRUE;
    CREATE RELATIONSHIP associated_tree(p,apex) WITHIN fsmtran;
    INSERT(associated_tree(p,apex), adissa);
  ELSE
     *** No previous tree. Mark the current one.
     root.mark <- TRUE;
     CREATE RELATIONSHIP associated_tree(p,root) WITHIN fsmtran;
     INSERT(associated_tree(p,root), adissa);
  ENDIF;
ENDIF;
```

```
        *** Final case.
        IF numdf = 1
          FIND as IN SET OF associated_tree WITHIN fsmtran WHERE as(p,*) AND
          as.top_root.mark = TRUE;
          *** We must find such relationship because c > 1.
          top <- as(p,?);
          top.mark <- FALSE;
          CREATE ENTITY tree (root) WITHIN fsmtran;
          INSERT(root,adissa);
          EXTERNAL assign_unique_name(root, fsmtran);
          root.mark <- TRUE;
          CREATE RELATIONSHIP connective(root,top,adf[1])
          (type <- "xor_type") WITHIN fsmtran;
          INSERT(connective(root,top,adf[1],adissa);
          CREATE RELATIONSHIP associated_tree(p,root) WITHIN fsmtran;
          INSERT(associated_tree(p,root),adissa);
        ENDIF;
        *** Destroy all the associated_tree that is not marked.
        FOR EACH as IN SET OF associated_tree WITHIN fsmtran WHERE as(p,*) AND
        as.top_root.mark = FALSE
          DESTROY RELATIONSHIP associated_tree (as) WITHIN fsmtran;
        ENDFOR;
      ENDIF;
    ENDFOR;
  ENDIF;

*** The rule takes an array of tree and join them using XOR.
RULE merge_tree(p: process, s: ARRAY OF tree, num: INTEGER;
                adissa: TRANS AGGREGATE OF
                        data_flow_aggregate);
  tree: top, root;
  data_flow_aggregate: fsmtran;
  INTEGER: i;

  fsmtran <- LOCATE_AGGREGATE(adissa);
  CREATE ENTITY tree (root) WITHIN fsmtran;
  EXTERNAL assign_unique_name(root,fsmtran);
  CREATE RELATIONSHIP connective(root,s[1],s[2]) (type <- "xor_type")
  WITHIN fsmtran;

  FOR i <- 3 TO num
    top <- root;
    CREATE ENTITY tree (root) WITHIN fsmtran;
    INSERT(root,adissa);
    EXTERNAL assign_unique_name(root, fsmtran);
    CREATE RELATIONSHIP connective(root,top,s[i])
    (type <- "xor_type") WITHIN fsmtran;
    INSERT(connective(root,top,s[i]),adissa);
  ENDFOR;

  CREATE RELATIONSHIP associated_tree(p,root) WITHIN fsmtran;
  INSERT(associated_tree(p,root),adissa);
  *** Mark the top root
  root.mark <- TRUE;

*** Merge all the terminator.
*** (1) No terminator in the ending set has outgoing flow.
```

```
*** (2) A terminator in the starting set has at most one outgoing flow.
RULE merge_terminators(adissa: TRAN AGGREGATE OF data_flow_aggregate);
  data_flow_aggregate: agg;
  terminator: t, copy_t;
  dataFlow: df, dfi, df_t;
  process: p;
  INTEGER: count;
  dataElement: de;

  agg <- LOCATE_AGGREGATE(adissa);
  FOR EACH t IN SET OF terminator WITHIN adissa WHERE
  t.belong_to_ending_set = TRUE
    FIND df IN SET OF dataFlow WITHIN adissa WHERE sends(t,df,*);
    IF NOT df = NULL
      *** t has at least one outgoing flow.
      *** Create a separate distinct occurrence.
      CREATE ENTITY terminator (copy_t) WITHIN agg;
      copy_t.belong_to_ending_set <- TRUE;
      EXTERNAL assign_unique_name(copy_t,agg);
      *** The copy has the same incoming flow element as the original
      *** but has no outgoing flow.
      FOR EACH df IN SET dataFlow WITHIN adissa WHERE sends(*,df,t);
        p <- sends(?,df,t);
        CREATE ENTITY dataFlow (copy_df) WITHIN agg;
        EXTERNAL assign_unique_name(copy_df,agg);
        FOR EACH de IN SET OF dataElement WITHIN agg WHERE
        dataElementsInFlow(df,de);
          CREATE RELATIONSHIP dataElementsInFlow(copy_df,de) WITHIN agg;
        ENDFOR;
        CREATE RELATIONSHIP sends(p,copy_df,copy_t);
        *** The copy has the same incoming flows as t.
        INSERT(sends(p,copy_df,copy_t),adissa);
      ENDFOR;
      t.belong_to_ending_set <- FALSE;
      t.belong_to_starting_set <- TRUE;
    ENDIF;
  ENDFOR:

  FOR EACH t IN SET OF terminator WITHIN adissa
  WHERE t.belong_to_starting_set = TRUE
    count <- 0;
    FOR EACH df IN SET OF dataFlow WITHIN adissa WHERE sends(t,df,*)
      count <- count+1;
    ENDFOR;
    IF count > 1
      *** t has more than one outgoing flow;
      FOR EACH df IN SET OF dataFlow WITIN adissa WHERE sends(t,df,*);
        *** For each outgoing data flow, create one separate copy
        *** which has the same incoming flow element as the original
        *** and has the corresponding outgoing data flow as the only
        *** outgoing flow.
        p <- sends(t,df,?);
        CREATE ENTITY terminator (copy_t) WITHIN agg;
        copy_t.belong_to_starting_set <- TRUE;
        EXTERNAL assign_unique_name(copy_t,agg);
        CREATE ENTITY dataFlow (copy_df) WITHIN agg;
        EXTERNAL assign_unique_name(copy_df,agg);
```

```
        FOR EACH de IN SET OF dataElement WITHIN agg
        WHERE dataElementsInFlow(df,de)
          CREATE RELATIONSHIP dataElementsInFlow(copy_df,de) WITHIN agg;
        ENDFOR;
        CREATE RELATIONSHIP sends(copy_t,copy_df,p);
        *** The copy has only one outgoing flow.
        INSERT(sends(copy_t,copy_df,p),adissa);
        FOR EACH dfi IN SET OF dataFlow WITHIN adissa WHERE sends(*,dfi,t)
          p <- sends(?,dfi,t);
          CREATE ENTITY dataFlow (copy_df) WITHIN agg;
          FOR EACH de IN SET OF dataElement WITHIN agg
          WHERE dataElementsInFlow(dfi,de);
            CREATE RELATIONSHIP dataElementsInFlow(copy_df,de);
          ENDFOR;
          EXTERNAL assign_unique_name(copy_df,agg)
          CREATE RELATIONSHIP sends(p,copy_df,copy_t) WITHIN adissa;
          *** The copy has the same incoming flows as t.
          INSERT(sends(p,copy_df,copy_t),adissa);
        ENDFOR;
      ENDFOR;
    ENDIF;
  ENDFOR;

*** To create the target finite state machines,
*** we traverse the FSM-transaction to trace the control flow.
*** We will first travel from each of the objects in the starting set,
*** and then we traverse from each of the CDFs that are involved in an AND.
RULE analyze_fsm_transaction(adissa: TRANS AGGREGATE OF
                                     data_flow_aggregate);
  INTEGER: i, count;
  dataFlow: df;
  SET OF dataFlow: b;
  terminator: t;
  fsm_aggregate: fsmagg;
  ARRAY OF fsm_aggregate: w;

  count <- 0;
  *** Assign a unique label to each CDF that is involved in an AND.
  FOR EACH df IN SET OF dataFlows WITHIN adissa
    WHERE df.involve = TRUE AND df.cdf = TRUE
    count <- count+1;
    df.p_num <- count;
    INSERT(df,b);
  ENDFOR;
  IF count = 0
    *** No ANDs
    CREATE AGGREGATE fsm_aggregate (fsmagg);
    ORACLE OUTPUT "Please supply a name for the finite state machine"
    ORACLE OUTPUT "which is generated from the aggregate"
    EXTERNAL display_trans_aggregate(adissa);
    ORACLE INPUT fsmagg.name;
    *** Generate the main FSM.
    generate_sequential_transaction(0,adissa,fsmagg);
  ELSE
    *** The transaction contains ANDs.
    CREATE AGGREGATE fsm_aggregate(fsmagg);
    ORACLE OUTPUT "Please supply a name for the finite state machine"
```

```
    ORACLE OUTPUT "which is generated from aggregate"
    EXTERNAL display_trans_aggregate(adissa);
    ORACLE INPUT fsmagg.name;
    *** Generate the main FSM.
    generate_sequential_transaction(0,adissa,fsmagg);
    i <- 1;
    FOR EACH df IN b
      CREATE AGGREGATE fsm_aggregate(w[i]);
      ORACLE OUTPUT "Please supply a name for the finite state machine"
      ORACLE OUTPUT "which is activated by the data flow"
      ORACLE OUTPUT df.name;
      ORACLE INPUT w[i].name;
      *** Generate the other FSMs.
      generate_sequential_transaction(df.p_num,adissa,w[i]);
      i <- i+1;
    ENDFOR;
    *** We need an external function to establish the relationship
    *** has_subtransaction. Each calling state will execute a bunch
    *** of FSMs depending on the boolean expression. Look at the RULE
    *** trace_path_with_process in which there is an explanation how
    *** the number of calling states are calculated.
    EXTERNAL link_fsm_aggregates(b,fsmagg,w);
  ENDIF;

*** This rule construct some necessary components of a FSM,
*** e.g., initial state, initial entry, predefined action, etc.
*** Moreover, based on the parameter num, we determine whether
*** we need to construct a transmission state.
RULE generate_sequential_transaction(num: INTEGER, adissa: TRANS AGGREGATE OF
                                     data_flow_aggregate, fsmagg:fsm_aggregate);
  dataFlow: df;
  terminator: t;
  state: initial_state, transmission_state;
  entry: en, initial_entry, empty_entry;
  action: ac;

  CREATE ENTITY state (initial_state) WITHIN fsmagg;
  initial_state.name <- "Initial state";
  initial_state.type_of_state <- "initial";
  IF num > 0
    initial_state.administrative_object <- TRUE;
  ELSE
    initial_state.administrative_object <- FALSE;
  ENDIF;

  CREATE ENTITY entry (initial_entry) WITHIN fsmagg;
  initial_entry.name <- "Initial entry";
  initial_entry.type_of_entry <- "initial";

  CREATE ENTITY empty (empty_entry) WITHIN fsmagg;
  empty_entry.name <- "empty entry";
  empty_entry.type_of_entry <- "empty";

  CREATE ENTITY action (ac) WITHIN fsmagg;
  ac.name <- "predefined action";

  CREATE RELATIONSHIP
```

```
  next_state_function(initial_state,initial_entry,initial_state,ac)
  WITHIN fsmagg;

  IF num = 0
    FOR EACH t IN SET OF terminator WITHIN adissa WHERE
    t.belong_to_starting_set = TRUE
      CREATE ENTITY action (ac) WITHIN fsmagg;
      ac.name <- "action of", t.name;
      INSERT(initial_state, MAP_OF(t) WITHIN fsmagg);
      INSERT(ac, MAP_OF(t) WITHIN fsmagg);
      FOR EACH df IN SET OF dataFlow WHERE sends(t,df,*) AND df.cdf = TRUE
        CREATE ENTITY entry (en) WITHIN fsmagg;
        en.name <- df.name;
        en.type_of_entry <- "intermediate";
        INSERT(en, MAP_OF(df) WITHIN fsmagg);
        *** We traverse the rest of the FSM-transaction via the CDF df.
        trace_path_with_dataflow(df, adissa, initial_state,fsmagg);
      ENDFOR;
    ENDFOR;
  ENDIF;
  *** When num is bigger than or equal to 1,
  *** the source aggregate does not contain any starting set.
  *** In this case, we create a transmission state.
  IF num > 0
    CREATE ENTITY state (transmission_state) WITHIN fsmagg;
    transmission_state.name <- "transmission state";
    transmission_state.type_of_state <- "intermediate";
    transmission_state.administrative_object <- TRUE;
    FIND df IN SET OF dataFlow WITHIN adissa WHERE df.p_num = num
    AND df.cdf = TRUE
    *** There is only one such data flow in this case.
    CREATE ENTITY entry (en) WITHIN fsmagg;
    en.name <- df.name;
    en.type_of_entry <- "intermediate";
    INSERT(en, MAP_OF(df) WITHIN fsmagg);
    CREATE ENTITY action (ac) WITHIN fsmagg;
    ac.name <- "transmission state action";
    CREATE RELATIONSHIP
      next_state_function(initial_state,en,transmission_state,ac)
      WITHIN fsmagg;
    trace_path_with_dataflow(df,adissa,transmission_state,fsmagg);
  ENDIF;

*** This rule takes a CDF (df), a FSM-transaction (adissa), a state (p_st)
*** and an FSM (fsmagg), traverses the FSM-transaction via the CDF.
*** The destination of the CDF will be mapped to a new state.
*** A state transition will be created linking p_st to the new state via
*** the image of df.
RULE trace_path_with_dataflow(df: dataFlow, adissa: TRANS AGGREGATE OF
                              data_flow_aggregate, p_st: state,
                              fsmagg: fsm_aggregate);
  process: p;
  terminator: t;
  dataStore: ds;
  state: st, inst;
  entry: en, ef;
```

```
FIND ef IN MAP_OF(df) WITHIN fsmagg WHERE TYPE_OF(ef) = entry;
FOR EACH p IN SET OF process WITHIN adissa WHERE sends(*,df,p)
  *** We reach a process.
  *** Avoid we have created the image of p already.
  IF IS_EMPTY(MAP_OF(p) WITHIN fsmagg) = TRUE
    CREATE ENTITY state (st) WITHIN fsmagg;
    st.name <- p.name;
    st.type_of_state <- "intermediate";
    st.administrative_object <- FALSE;
    CREATE ENTITY action (ac) WITHIN fsmagg;
    ac.name <- "action of ", p.name;
    CREATE RELATIONSHIP next_state_function(p_st,ef,st,ac) WITHIN fsmagg;
    *** Map the process to the state and the action element.
    INSERT(st, MAP_OF(p) WITHIN fsmagg);
    INSERT(ac, MAP_OF(p) WITHIN fsmagg);
    *** Continue to traverse.
    trace_path_with_process(p,adissa,st,fsmagg);
  ENDIF;
ENDFOR;

*** Locate the initial state.
FIND inst IN SET OF state WITHIN fsmagg
  WHERE st.type_of_state = "initial";

FOR EACH t IN SET OF terminator WITHIN adissa WHERE sends(*,df,t)
  *** We reach a terminator.
  IF IS_EMPTY(MAP_OF(t) WITHIN fsmagg) = TRUE
    *** The image set is empty. That means t has not been mapped to anything.
    *** It cannot be in the starting set because every terminator
    *** in the starting set has been mapped to a state already.
    CREATE ENTITY state (st) WITHIN fsmagg;
    st.type_of_state <- "final";
    st.administrative_object <- FALSE;
    st.name <- t.name;
    CREATE ENTITY action (ac) WITHIN fsmagg;
    ac.name <- "action of ", t.name;
    CREATE RELATIONSHIP next_state_function(p_st,ef,st,ac) WITHIN fsmagg;
    INSERT(st, MAP_OF(t) WITHIN fsmagg);
    INSERT(ac, MAP_OF(t) WITHIN fsmagg);
  ENDIF;

  IF SIZE_OF(MAP_OF(t) WITHIN fsmagg) > 0
    *** The image of t is not empty. It could means
    *** (a) the image of t has been created in the current rule.
    ***     We just come back to this rule because the existence
    ***     of some other path which leads to the terminator.
    ***     In this case, we should not create the image of the
    ***     terminator twice.
    *** (b) t is in the starting set and is mapped to the initial
    ***     state.
    IF inst IN MAP_OF(t) WITHIN fsmagg
      *** Case (b).
      FIND ac IN MAP_OF(t) WITHIN fsmagg WITHIN fsmagg
      WHERE TYPE_OF(ac) = action;
      *** Find the action element of the terminator.
      CREATE RELATIONSHIP next_state_function(p_st,ef,inst,ac) WITHIN fsmagg;
    ENDIF;
```

```
  ENDIF;
 ENDFOR;

 *** We reach a data store.
 FOR EACH ds IN SET OF dataStore WITHIN adissa WHERE changes(*,df,ds)
 AND df.cdf = TRUE
   *** Avoid creating the image of ds twice.
   IF IS_EMPTY(MAP_OF(ds) WITHIN fsmagg) = TRUE
     CREATE ENTITY state (st) WITHIN fsmagg;
     st.name <- df.name;
     st.type_of_state <- "final";
     st.administrative_object <- FALSE;
     CREATE ENTITY action (ac) WITHIN fsmagg;
     ac.name <- "action of ", ds.name;
     INSERT(st, MAP_OF(ds) WITHIN fsmagg);
     INSERT(ac, MAP_OF(ds) WITHIN fsmagg);
     CREATE RELATIONSHIP next_state_function(p_st,ef,st,ac) WITHIN fsmagg;
   ENDIF;
 ENDFOR;

*** This rule takes a process (p), an FSM-transaction (adissa), a state
*** (p_st) and an FSM (fsmagg), determine whether a calling process
*** should be created by examining all the outgoing CDFs of p.
*** If some CDFs are not involved in AND-relationships, we traverse the
*** FSM-transaction via those CDFs. In this case, we invoke the
*** previous rule and pass the necessary parameter to it.
RULE trace_path_with_process(p: process, adissa: TRANS AGGREGATE OF
                             data_flow_aggregate, p_st: state,
                             fsmagg: fsm_aggregate);
  dataFlow: df, sdf;
  process: q;
  terminator: t;
  tree: top;
  entry: en;
  state: calling_state, exit_state;
  BOOLEAN: tag;
  INTEGER: count, i;
  action: ac;

  tag <- FALSE;
  FOR EACH df IN SET OF dataFlows WITHIN adissa WHERE
  df.cdf = TRUE AND sends(p,df,*)
    IF IS_EMPTY(MAP_OF(df) WITHIN fsmagg) = TRUE
      *** Avoid we have created the image already.
      IF df.involve = FALSE
        CREATE ENTITY entry (en) WITHIN fsmagg;
        en.name <- df.name;
        en.type_of_entry <- "intermediate";
        INSERT(en, MAP_OF(df) WITHIN fsmagg);
        *** Continue to trace the FSM-transaction.
        trace_path_with_dataflow(df,adissa,p_st,fsmagg);
      ELSE
        tag <- TRUE;
      ENDIF;
    ENDIF;
  ENDFOR;
```

```
*** If some of the outgoing flows of p are involved
*** in AND, we need to create some calling states.
IF tag = TRUE
  *** Find out how many calling functions we have to create.
  *** We need an external function which takes a boolean expression
  *** ex1 XOR ex2 XOR ... XOR exn
  *** where each ex a conjunction of data flows or a single data flow.
  *** One calling function is need for each ex if
  *** (1) ex is a conjunction.
  *** (2) ex is a single data flow but is involved in an AND.
  top <- associated_tree(p,?);
  count <- EXTERNAL calculate_num_of_calling_function(top,adissa);
  FOR i <- 1 TO count
    CREATE ENTITY state (calling_state) WITHIN fsmagg;
    EXTERNAL assign_unique_name(calling_state,fsmagg);
    calling_state.type_of_state <- "intermediate";
    calling_state.administrative_object <- TRUE;
    CREATE ENTITY entry (en) WITHIN fsmagg;
    EXTERNAL assign_unique_name (en, fsmagg);
    en.type_of_entry <- "intermediate";
    CREATE ENTITY action (ac) WITHIN fsmagg;
    ac.name <- "action of ", calling_state.name;
    CREATE RELATIONSHIP next_state_function(p_st,en,calling_state,ac)
      WITHIN fsmagg;
    CREATE ENTITY state (exit_state) WITHIN fsmagg;
    EXTERNAL assign_unique_name(exit_state, fsmagg);
    exit_state.type_of_state <- "final";
    exit_state.administrative_object <- TRUE;
    CREATE ENTITY entry (en) WITHIN fsmagg;
    EXTERNAL assign_unique_name (en, fsmagg);
    en.type_of_entry <- "intermediate";
    CREATE ENTITY action (ac) WITHIN fsmagg;
    ac.name <- "action of ", exit_state.name;
    CREATE RELATIONSHIP
      next_state_function(calling_state,en,exit_state,ac) WITHIN fsmagg;
  ENDFOR;
ENDIF;
```